(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
17 January 2002 (17.01.2002)

PCT

(10) International Publication Number
WO 02/05140 A1

(51) International Patent Classification[7]: G06F 17/30, 15/16, H04B 17/10

(21) International Application Number: PCT/US01/21829

(22) International Filing Date: 10 July 2001 (10.07.2001)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
60/217,594 11 July 2000 (11.07.2000) US
09/709,234 9 November 2000 (09.11.2000) US

(71) Applicant *(for all designated States except US)*: **LAUNCH MEDIA, INC.** [US/US]; Attn: Legal Department, 2700 Pennsylvania Ave., Santa Monica, CA 90404 (US).

(72) Inventors; and

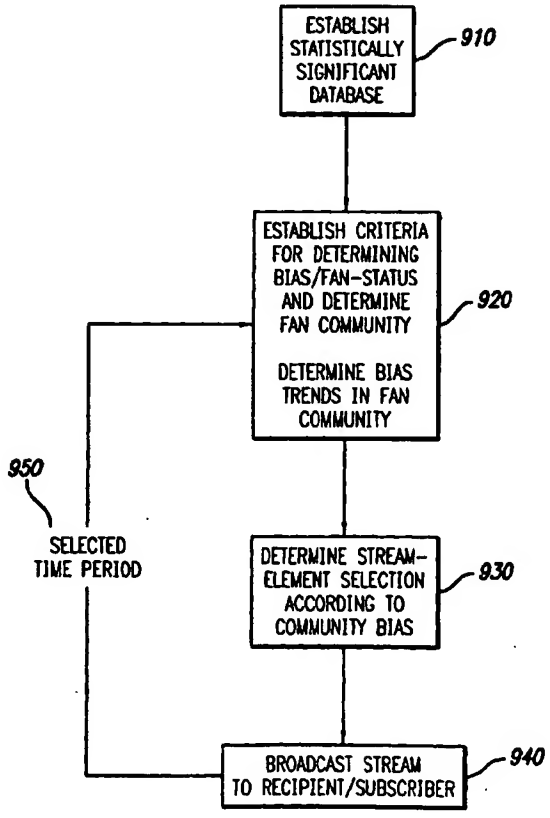
(75) Inventors/Applicants *(for US only)*: **BOULTER, Jeffrey** [US/US]; LAUNCH Media, Inc., Attn: Legal Dept., 2700 Pennsylvania Ave., Santa Monica, CA 90404 (US). **BEAUPRE, Todd** [US/US]; LAUNCH Media, Inc., Attn: Legal Dept., 2700 Pennsylvania Ave., Santa Monica, CA 90404 (US). **VEILLEUX, John-Paul** [US/US]; LAUNCH Media, Inc., Attn: Legal Dept., 2700 Pennsylvania Ave., Santa Monica, CA 90404 (US).

(74) Agents: **JORDAN, Andrew** et al.; Cislo & Thomas LLP, 233 Wilshire Blvd., Ste. 900, Santa Monica, CA 90401-1211 (US).

(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW,

[Continued on next page]

(54) Title: ONLINE PLAYBACK SYSTEM WITH COMMUNITY BIAS



(57) Abstract: A method (900) for entertaining individuals according to a community having similar tastes. Information derived from user accounts (910) form the basis of a community (920) and collateral preferences allow other subscribing individuals to enjoy the benefit of wider-ranging tastes according to the preferences expressed by the other members of the community. Additionally, assuming that individuals sharing one preference in common may be likely to share others, the present method allows those who choose to listen to the "fan station" the ability to enjoy similar music or other data streams according to preferences expressed by the fan community as a whole (930).

WO 02/05140 A1

MX, MZ, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, US, UZ, VN, YU, ZA, ZW.

(84) **Designated States** *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published:
— *with international search report*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

ONLINE PLAYBACK SYSTEM WITH COMMUNITY BIAS

TECHNICAL FIELD

This invention relates to database generation and data stream transmission, and more particularly to biased data stream transmission method according to a community of subscribers or fans enjoying similar tastes.

BACKGROUND ART

In an online environment, the demand for digital entertainment is limited by statute in the United States of America under the Digital Millennium Copyright Act (DMCA, Digital Millennium Copyright Act of 1998, Public Law 105-304). Legitimate providers of online entertainment must adhere to the DMCA and pay license fees for the copyrighted works broadcast over the Internet or other online environment. Otherwise, such providers are liable for copyright infringement.

The Digital Millennium Copyright Act (DMCA) addresses protections for copyrighted works transmitted online. The DMCA entitles websites that stream music to a statutory license to perform copyrighted sound recordings as long as they meet certain requirements. Compliance with these requirements by, among other ways: not streaming over a three-hour period, more than three songs or more than two in a row from the same recording, or four songs or more than three in a row from the same recording artist or anthology; and by transmitting songs in a noninteractive format by, for example, not allowing users to specifically create or request programming on demand or to hear programming at designated times. Additionally, compliance with the DMCA requires that advance song or artist playlists not be published.

In an online environment, the content provider may "narrowcast" the data feed to a single individual and still comply with the DMCA even though thousands of individual narrowcast transmissions are made simultaneously. For example, so long as each individual narrowcast does not violate the DMCA, compliance with the DMCA is maintained.

"Narrowcasting" is a term that may be new in the art. As a contrast to "broadcasting" where information is broadcast on a wide basis and generally available to anyone with a tuned receiver, "narrowcasting" arises from the individually addressable data packets used in TCP/IP protocol. The packets are addressed to individual computers and include almost all forms of data transmission over the Internet. Consequently, when broadcasting occurs on the Internet, it is generally composed of a bundle of narrowcast packets as each one must be individually addressed to the computers of the audience. This is true even though several computers are receiving the same content at the same time. Each computer must be individually addressed even though the packets are identical. When demand is high for Internet content such as a live performance or transmission, bandwidth may not be sufficient for all who request transmission.

Due to the nature of Internet communications and TCP/IP protocol, narrowcasting is one of the basic and easy ways in which to transmit information packets. Multicasting may also be used (See Bob Quinn, Killer Network Apps That Aren't Network Killers, Dr. Dobb's Journal October 1997), but has drawbacks due to technical obstacles in effecting a multicast on the open Internet. Other protocols (such as FTP) also exist.

Under the LAUNCHcast™ system (the subject of PCT International Application number WO 01/35667 A1 which is incorporated herein by this reference thereto), each subscriber may "tune" his or her narrowcast by expressing preferences that are recorded and preserved in an account associated with the user/subscriber.

The LAUNCHcast™ system provides a means by which DMCA compliance can be maintained while biasing narrowcast transmissions according to audience/individual preferences. By soliciting, receiving, and recording an individual's preferences regarding (for example) a music data stream, LAUNCH Media, Inc. provides digital audio feed to a subscriber that both complies with the DMCA as well as catering to the individual's musical tastes. If the musical tastes of the individual are limited, additional music may be used to fill in "airtime" or "nettime" that cannot be filled with the individual's favorite songs as such transmission would violate the DMCA. Conversely, an individual with broad tastes could have very few works transmitted in the data stream that fall outside of the individual's tastes.

Very often, people who enjoy one type of music or artist also enjoy other types of music or artists so that an appearance of association between the two occur without an obvious causal link. For example, individuals who enjoy music by Barry Manilow might also enjoy the music of Neil Sedaka in a high percentage that may exceed random statistical occurrence. Consequently, when accompanied by a rating system or engine, individuals who enjoy Barry Manilow might welcome music by Neil Sedaka although they may have never heard music by Neil Sedaka before.

The present invention allows enhancement of narrowcast transmission for the listener's or consumer's enjoyment while maintaining compliance with the DMCA. By associating communities of listeners/consumers around specific artists or genres, subscribers or listeners of an online data stream entertainment service are provided with a more focused and enjoyable experience as the data stream is catered to their preference by using a community bias based upon those who enjoy such artists, an individual artist, genres, or an individual genre.

Note should be taken that the method described herein pertains not only to audio data streams, but any sort of data stream where preferences may be present, including video and multimedia. As entertainment data streams are particularly susceptible to strong personal preferences, the present invention resolves a need for providing dynamic accommodation of expressed preferences in a community of subscribers or listeners while complying with applicable copyright law.

Articles addressing multicasting include:

Indira Gandhi National Open University, Comparative analysis - TCP - UDP, published on the Internet at http://www.laynetworks.com/users/webs/tcp_udp.htm; and

Bob Quinn, Killer Network Apps That Aren't Network Killers, Dr. Dobb's Journal October 1997 also published on the Internet at http://www.ddj.com/articles/1997/9710/9710b/9710b.htm.

DISCLOSURE OF INVENTION

The present invention provides for a biased data stream that is biased according to those who prefer data streams of particular types. Using the example above, a community enjoying Barry Manilow could be used to bias a data stream towards both songs by Barry Manilow and those songs that the members of the Barry Manilow community enjoy. Consequently, through the use of preferences expressed by feedback of each individual member of the Barry Manilow community, a Barry Manilow-based radio station or data stream set emerges. Narrowcasting based upon such a biased data stream may then be subject to DMCA constraints so that no one narrowcast transmission violates the DMCA, yet the data stream transmission is biased according preferences expressed by the Barry Manilow community as a whole.

The biasing of such a data stream becomes more robust and more reliable with greater numbers of members and when such members express a large number of preferences regarding the type of music they enjoy.

Note should be taken that the term "music" as used herein is used as a shorthand for any data stream subject to taste or preference. Music data streams form a basic analogy from which all other data streams may be comparably

likened, unless otherwise indicated. Additionally, the use of Barry Manilow as an artist of preference is arbitrary and could be substituted by current, modern, or classical artists such as Melissa Etheridge, Karen Carpenter, Rosemary Clooney, Phil Harris, Hank Williams, Led Zeppelin, Luciano Pavarotti, or Spike Jones.

BRIEF DESCRIPTION OF DRAWINGS

Figure 1 shows an exemplary page for an artist, in this case Tori Amos.

Figure 2 shows a similar exemplary artist page with the Fan Station option highlighted.

Figure 3 is an isolated view of the Fan Station option shown in Figures 1 and 2.

Figure 4 is an enlarged view of Figure 3.

Figure 5 shows an alternative exemplary page for an artist, in this case Tori Amos.

Figure 6 shows a similar alternative exemplary artist page with the Fan Station option highlighted.

Figure 7 is an isolated view of the alternative Fan Station option shown in Figures 5 and 6.

Figure 8 is an enlarged view of Figure 7.

Figure 9 is a diagrammatic view of steps taken in the present invention

MODE(S) FOR CARRYING OUT THE INVENTION

The detailed description set forth below in connection with the appended drawings is intended as a description of presently-preferred embodiments of the invention and is not intended to represent the only forms in which the present invention may be constructed and/or utilized. The description sets forth the functions and the sequence of steps for constructing and operating the invention in connection with the illustrated embodiments. However, it is to be understood that the same or equivalent functions and sequences may be accomplished by different embodiments that are also intended to be encompassed within the spirit and scope of the invention.

The present invention resides in the establishment of a community based upon shared musical tastes. Upon receiving and recording a statistically significant number of preferences and feedback regarding songs, those who prefer an artist may be distinguished from other users who may form a background against which fans of such an artist are distinguished.

Using as an example the contemporary artist Tori Amos, Figures 1 – 8 show alternative commercial presentations of the present invention. As for almost all artists in its library, LAUNCH Media maintains home pages for artists from which users/subscribers may select links to additional information, including the purchase of works by the artist. As an option on the home page, interested individuals may select to hear an audio stream based upon the preferences of users who like that artist, in this case, who like Tori Amos.

By selecting the "listen" or "watch" links in the Fan Station section of the Tori Amos home page (Figures 3 and 4 and Figures 7 and 8), individuals can receive data streams biased according to a community that likes Tori Amos. As the Tori Amos community may tend to share other musical tastes, the data stream that results from the Fan Station link selection may also entertain the individual so selecting the link as that individual's tastes may correspond to the tastes of the Tori Amos community as a whole just as it did with the artist Tori Amos.

In order to determine a community's preferences, only those individuals in the subscriber database who are "fans" of the artist are used to determine the community's preferences. The term "fan" may be arbitrarily defined as those individual subscribers who rate Tori Amos as a 70 or more on a scale of 100 with 0 being a least favorite artist

and 100 being a most favorite artist. The choice is arbitrary but needs to reflect a bias sufficient to entertain, or even delight, those who choose to listen to the community channel.

Upon determining the community of interest (Tori Amos fans, for example), collateral data regarding other preferences are gathered from those same individuals who are designated fans of Tori Amos. For example, in one embodiment, for each member of the community, all other rated artists besides Tori Amos are inspected. Those artists who also scored 70 or higher are noted and temporarily stored in a database. After all of the member accounts of the community have been polled, those artists who are present in 70% of the accounts may be chosen as artists whose music will also be transmitted as secondary musical selections in narrowcast to those who choose the Tori Amos Fan Station.

In an alternative embodiment, the collateral artists may be chosen according to popularity with no floor threshold (of 70% as in the embodiment above, or otherwise). In another alternative embodiment, songs rated by the community may take precedent over artist ratings such that individual songs are selected for narrowcast transmission from community preferences as opposed to portfolios of songs according to different artists (again according to community preferences).

In this way, a community may be defined and its preferences determined. Of course, other data streams subject to preference or taste may be substituted for the music/audio data stream as set forth in the example above, including video, multimedia, or otherwise.

The present invention is shown diagrammatically in Figure 9. As shown in Figure 9, the present invention **900** provides steps for achieving the community bias system in order to provide data streams consistent with such community preferences. The online playback system with community bias **900** of the present invention begins first with establishing a statistically-significant database **910**. This database may be a database comprised of all users of a system such as LAUNCHcast™ or the like. Such a statistically-significant database has entries with artistic preferences of the individual subscribers. Such preferences may include artists and songs preferred and not preferred (liked and disliked), as well as albums that the subscribers or recipients prefer or do not prefer.

The entire subscriber community generally defines the artistic or preferential "space" in which the present invention operates. Using such a geometrical point of view, certain sub-areas of the artistic database may then be the subject of the community preference system **900** set forth herein. In order to achieve the present invention, certain delimitations must be made as to what defines a community, and the preferences expressed by the subscriber/recipient serve in this capacity.

Statistical significance is a relative term. The goal of the present invention is to provide entertainment or other desired data streams to the recipients. Here, the data streams are songs or music videos. However, other data streams subject to subscriber databases where preferences are expressed for the content or type of data stream may also put to good use the present invention and are within the scope of the present invention and of the claims set forth herein. Statistical significance arises in the form of certain threshold criteria by which certain preferences are deliminated and/or distinguished from others. Generally, those who listen to country music may not want to also listen to heavy metal music. Those who would prefer rap may also like to listen to hip-hop music. Those who enjoy classical music may not enjoy swing or polka music. Depending upon the available databases of both subscribers and data streams, certain subgenres may be available such as all-Mozart or all-Beethoven community channels.

While feedback may be obtained from the recipients of the community-biased data streams, generally the present invention uses the rule of thumb of approximately "70" as the rating threshold by which a person is considered to be a "fan" of the artist or the like. The "70" rating could be interpreted as indicating that the artist is in the top one-third

(1/3) of the individual's preferred artists. By dwelling in this top 1/3 area, a community may be defined, although the exact numerical criteria may depend upon the range of the "space" available for use in the present invention, as well as the number of subscribers and data streams. Generally, the broader and more numerous the original and primary database of subscribers and datastreams, the higher and more exclusive the threshold rating may be.

Upon establishing a statistically-significantly database **910**, certain criteria must be established for determining a community's bias **920**. Upon choosing that threshold, the statistically-significant database **910** is then filtered, sorted, or evaluated, to determine what trends are present with respect to fan or subscriber preferences. As mentioned above, the rating of an artist of approximately above 70 on a scale of 0 – 100 is considered to be a relevant and significant threshold. The use of artists to define a genre or a consistent theme with respect to music generally arises from the fact that artists tend to write the same kind of music or the same type of music much in the same way as Vivaldi and Mozart had their own separate and distinct styles.

Upon determining the trends in the fan community **920**, a selection of individual stream elements may be made **930**. Such stream elements are generally in conformance with two criteria: the community bias trends established in step **920**, as well as any applicable copyright law. In the United States, the Digital Millenium Copyright Act (DMCA) generally controls such on-line transmissions of copyright works such as sound recordings and audio-visual works.

The stream selection step **930** may be achieved in two modes of transmission. One mode would be a narrowcast mode where different individual streams are transmitted to different recipients who have chosen and are currently listening to a community fan station. Alternatively, one transmission stream could be distributed simultaneously to all current listeners of the fan station/community channel. Both of these transmission methods are in compliance with the DMCA and provide alternative means by which the present invention **900** may be realized.

When an individual hears a song on a community channel that he or she ("he") would like to rate, the rating tool may be made available to him via the player tool. The rating so made by the individual is then recorded in his or her preferential settings as a subscriber to the database **910**. The user's ratings may indirectly affect the data stream selection **930** as it may form part of the database used to determine the community and the stream selected for the corresponding channel. The user must be a fan of the artist, for example, to effect that artist's community channel.

Once the stream selection process **930** has been performed, the stream is then broadcast to the recipient(s)/subscriber(s) **940**. The recipients then enjoy the receipt of the data streams and may be exposed to new music according to their own expressed preference indicated by subscribing to the fan station. Consequently, an individual who likes country music and chooses a Hank Williams community channel may be exposed to music by Porter Wagner which he or she may also like. The same is similarly true for contemporary musical style such as rap and hip-hop, as well as musical styles developed in the past, currently under development, or to be developed in the future.

In order to maintain the relevance of the community channel/fan station, the trend determining step **920**, stream selection step **930** may be re-engaged after a certain period of time ranging from one week to several months **950**. This allows for those who enjoy a certain type of music to benefit from currently-popular related styles and to allow the stream selection process **930** to be updated to reflect current tastes.

While the present invention has emphasized entertainment in the form of data streams relating to songs, sound recordings, and audio visual work such as music videos, the present invention is also applicable to data stream transmission systems that must comply with a regulatory scheme (such as the DMCA) in view of express preferences for content and/or type (such as the music individual persons like and dislike). Certain automated processes may benefit from the present invention, as machine-implemented processes may operate under a wide variety of conditions and

benefit from the transmission of data streams such as information- and/or content-dependent data streams dependent upon a wide variety of factors, including geographic location, climate, other environmental conditions, or otherwise. For example, the data streams may be sets of suggested instructions for artificially-intelligent systems operating under situations requiring problem-solving abilities.

The source code listing sets forth with particularity certain software methods by which one embodiment of the present invention may be achieved. The listing is believed to provide a full and complete disclosure of one embodiment of the present invention.

While the present invention has been described with reference to a preferred embodiment or to particular embodiments, it will be understood that various changes and additional variations may be made and equivalents may be substituted for elements thereof without departing from the scope of the invention or the inventive concept thereof. In addition, many modifications may be made to adapt a particular situation or material to the teachings of the invention without departing from the essential scope thereof. Therefore, it is intended that the invention not be limited to particular embodiments disclosed herein for carrying it out, but that the invention includes all embodiments falling within the scope of the appended claims.

INDUSTRIAL APPLICABILITY

It is an object of the present invention to provide more entertaining online data feeds.

It is another object of the present to provide more entertaining data streams by providing a biased data stream according to a listener's/consumer's preferences.

It is yet another object of the present invention to provide a more entertaining data stream by biasing a data stream according to a community expressing preferences for significant components of the data stream, such as an artist or genre.

It is yet another object of the present invention to provide a community biased music data stream according to a community expressing preferences for music carried by said data stream, such as an artist or genre.

These and other objects and advantages of the present invention will be apparent from a review of the following specification and accompanying drawings.

TEXT LISTING OF SOURCE CODE

The following copyrighted source code provides a realizable embodiment of the present invention and is presented by way of example and not by limitation. Other source code and compilations thereof may implement the present invention without duplicating the following source code.

```
package com.launch.rm.lc.SimilaritiesEngine;

import java.util.Hashtable;
import java.util.Enumeration;
import java.util.Vector;

/**
 * This class finds a bunch of items that a group of users have
 * in common that they've rated highly. The items are sorted from
 * highest group rating to lowest group rating.
```

```
*
* @author Jeff Boulter
* @author John Veilleux
*/
public class Consensus
{
	private Hashtable contenders = new Hashtable();
	private Vector finalistIDVec = new Vector();
	private int itemID = -1;
	private int ratingCount = 0;


	/**
	 * Creates an empty consensus.
	 */
	public Consensus()
	{
	}

	/**
	 * Creates a consensus with an item that should be excluded from
	 * the users' ratings.
	 *
	 * @param	itemID		the ID of the item to exclude
	 */
	public Consensus( int itemID )
	{
		this.itemID = itemID;
	}

	/**
	 * Creates a consensus where the list of items generated doesn't
	 * have to exclude a specific item.
	 *
	 * @param	userRatings			the user ratings
	 */
	public Consensus( Vector userRatings )
	{
		addRatings( userRatings );
	}

	/**
	 * Creates a consensus where the given item must be excluded
	 * from the list of items that's generated.
	 *
	 * @param	itemID		the ID of the item to exclude
	 * @param	userRatings	the user ratings
	 */
	public Consensus( int itemID, Vector userRatings )
	{
		this.itemID = itemID;

		addRatings( userRatings );
	}

	/**
	 * Polls the group of users for their final list of items.
	 *
	 * @return	the list of item ID's ordered from highest to lowest
	 *			group rating
```

```
 */
public OrderedList poll()
{
        OrderedList result = new OrderedList();
        Integer ratingItemID = null;
        GroupRating groupRating = null;

        for ( int i = finalistIDVec.size() - 1; i >= 0; i -- )
        {
                ratingItemID = (Integer) finalistIDVec.elementAt( i );
                groupRating = (GroupRating) contenders.get( ratingItemID );

                result.add( groupRating.get(), groupRating );
        }

        return result;
}

/**
 * Gets the total number of ratings within the pool of users.
 *
 * @return      the rating count
 */
public int getRatingCount()
{
        return ratingCount;
}

/**
 * Adds ratings to the consensus by users who will determine the
 * final list of items.
 *
 * @param       userRatings             the vector containing each user's
 *                                                      ratings
 */
public void addRatings( Vector userRatings )
{
        Rating r;

        r = null;

        for ( int i = userRatings.size() - 1; i >= 0; i -- )
        {
                r = (Rating) userRatings.elementAt( i );

                if ( r.itemID != itemID )
                {
                        add( r );
                }
        }
}

/**
 * Adds a rating to be used in the calculation of a contender
 * item's group rating. Once an item gets a specified minimum
 * number of ratings to calculate a group rating, it gets put
 * into the finalist list.
 */
private void add( Rating r )
{
```

```
                Integer ratingItemID = new Integer( r.itemID );
                GroupRating contenderGR = (GroupRating) contenders.get( ratingItemID );

                if ( contenderGR == null )
                {
                        contenderGR = new GroupRating( r.itemID );

                        contenders.put( ratingItemID, contenderGR );
                }
                else if ( contenderGR.getNumRatings() == (
SimilaritiesConstants.MIN_FANS_FOR_RECOMMENDED_ITEM - 1 ) )
                {
                        finalistIDVec.addElement( ratingItemID );
                }

                ratingCount ++;
                contenderGR.add( r );
        }
}
```

```
package com.launch.rm.lc.SimilaritiesEngine;

import com.launch.rm.lc.PlaylistGenerator.Constants;
import java.util.Vector;
import java.io.*;
import java.sql.ResultSet;


/**
 * This class generates a file containing items and their similar
 * items. This is for debug purposes only; components used in the
 * calculations of the similarities are printed out.
 *
 * @author        John Veilleux
 */
public class DataFileGenerator
{
        private final static int MAX_ITEMS_TO_WRITE = 100;
        private final static int MAX_SIMILAR_ITEMS_PER_ITEM = 10;
        private final static String OUTPUT_FILENAME_ROOT = "\\export\\";
        private final static String OUTPUT_FILENAME_SUFFIX = "Similarities.txt";
        private final static String TYPE_STRING[] = { "Artist", "Song", "Album", "Artist" };

        static
        {
                System.setErr( System.out );
        }


        public static void main( String args[] )
        {
                Integer itemID;
                Byte itemType;
                Byte ratingType;
                SimilaritiesEngine engine;
                Vector itemIDVec;
                OrderedList groupRatingList;
                ResultSet rs;
                PrintWriter writer;
                GroupRating gRating;
                double gRatingValue;
                String headerStr1;
                String headerStr2;
                String itemSQL;
                String ratingSQL;
                String itemParamStr1;
                String itemParamStr2;
                String ratingParamStr1;
                String ratingParamStr2;

                itemID = null;
                itemType = null;
                ratingType = null;
                engine = null;
                itemIDVec = new Vector();
                groupRatingList = null;
                rs = null;
                writer = null;
                gRating = null;
                gRatingValue = 0;
```

```
headerStr1 = null;
headerStr2 = null;
itemSQL = null;
ratingSQL = null;
itemParamStr1 = null;
itemParamStr2 = null;
ratingParamStr1 = null;
ratingParamStr2 = null;

try
{
	switch ( args.length )
	{
		case 2
		:
			ratingType = new Byte( args[ 1 ] );

			if ( ratingType.byteValue() < Constants.ITEM_TYPE_SONG ||
ratingType.byteValue() > Constants.ITEM_TYPE_ARTIST )
			{
				throw new Exception( "Rating type must be " +
Constants.ITEM_TYPE_SONG + ", "+ Constants.ITEM_TYPE_ALBUM + ", or " +
Constants.ITEM_TYPE_ARTIST + "." );
			}

		case 1
		:
			itemType = new Byte( args[ 0 ] );

			if ( itemType.byteValue() < Constants.ITEM_TYPE_SONG ||
itemType.byteValue() > Constants.ITEM_TYPE_ARTIST )
			{
				throw new Exception( "Item type must be " +
Constants.ITEM_TYPE_SONG + ", " + Constants.ITEM_TYPE_ALBUM + ", or " +
Constants.ITEM_TYPE_ARTIST + "." );
			}

			break;

		default
		:
			throw new InstantiationException();
	}

	if ( ratingType != null && itemType.byteValue() == ratingType.byteValue() )
	{
		throw new Exception( "Item type cannot be equal to rating type." );
	}

	Debugger.out( "DataFileGenerator started." );
	Debugger.resetTimer( "DataFileGenerator" );

	switch ( itemType.intValue() )
	{
		case Constants.ITEM_TYPE_SONG
		:
			itemSQL = "exec sp_lcGetSongDetail_xsxx ";
			itemParamStr1 = "title";
			itemParamStr2 = "artist";
```

```
                    break;

                case Constants.ITEM_TYPE_ALBUM
                :
                    itemSQL = "exec sp_lcGetAlbumDetail_xsxx ";
                    itemParamStr1 = "albumName";
                    itemParamStr2 = "artistName";

                    break;

                case Constants.ITEM_TYPE_ARTIST
                :
                    itemSQL = "exec sp_lcGetArtistInfo_xsxx ";
                    itemParamStr1 = "artist";

                    break;
            }

            if ( ratingType == null )
            {
                engine = new SimilaritiesEngine( itemType.byteValue(),
MAX_ITEMS_TO_WRITE );
                writer = new PrintWriter( new FileWriter( OUTPUT_FILENAME_ROOT +
TYPE_STRING[ itemType.intValue() ] + OUTPUT_FILENAME_SUFFIX ) );
                headerStr1 = TYPE_STRING[ itemType.intValue() ] + "s similar to (";
                ratingSQL = itemSQL;
                ratingParamStr1 = itemParamStr1;
                ratingParamStr2 = itemParamStr2;
            }
            else
            {
                engine = new SimilaritiesEngine( itemType.byteValue(),
ratingType.byteValue(), MAX_ITEMS_TO_WRITE );
                writer = new PrintWriter( new FileWriter( OUTPUT_FILENAME_ROOT +
TYPE_STRING[ itemType.intValue() ] + TYPE_STRING[ ratingType.intValue() ] +
OUTPUT_FILENAME_SUFFIX ) );
                headerStr1 = TYPE_STRING[ ratingType.intValue() ] + "s similar to " +
TYPE_STRING[ itemType.intValue() ] + " (";

                switch ( ratingType.intValue() )
                {
                    case Constants.ITEM_TYPE_SONG
                    :
                        ratingSQL = "exec sp_lcGetSongDetail_xsxx ";
                        ratingParamStr1 = "title";
                        ratingParamStr2 = "artist";

                        break;

                    case Constants.ITEM_TYPE_ALBUM
                    :
                        ratingSQL = "exec sp_lcGetAlbumDetail_xsxx ";
                        ratingParamStr1 = "albumName";
                        ratingParamStr2 = "artistName";

                        break;

                    case Constants.ITEM_TYPE_ARTIST
                    :
                        ratingSQL = "exec sp_lcGetArtistInfo_xsxx ";
```

```
                                        ratingParamStr1 = "artist";

                                        break;
                                }
                        }

                        itemIDVec = engine.getItemIDs();

                        for ( int i = 0; i < itemIDVec.size(); i ++ )
                        {
                                itemID = (Integer) itemIDVec.elementAt( i );
                                headerStr2 = headerStr1 + itemID + ") ";
                                rs = DBConnection.executeSQL( itemSQL + itemID, false );

                                if ( rs.next() )
                                {
                                        headerStr2 += rs.getString( itemParamStr1 );

                                        if ( itemParamStr2 != null )
                                        {
                                                headerStr2 += " by " + rs.getString( itemParamStr2 );
                                        }
                                }

                                rs.close();

                                writer.println( headerStr2 );

                                groupRatingList = engine.getSimilar( itemID,
MAX_SIMILAR_ITEMS_PER_ITEM );

                                for ( int j = 0; j < groupRatingList.size(); j ++ )
                                {
                                        gRating = (GroupRating) groupRatingList.elementAt( j );
                                        gRatingValue = groupRatingList.valueAt( j );

                                        writer.print( "\t" + gRating.toBigString() + "\t" );

                                        rs = DBConnection.executeSQL( ratingSQL + gRating, false );

                                        if ( rs.next() )
                                        {
                                                writer.print( rs.getString( ratingParamStr1 ) );

                                                if ( ratingParamStr2 != null )
                                                {
                                                        writer.print( "\t" + rs.getString( ratingParamStr2 ) );
                                                }
                                        }

                                        rs.close();

                                        writer.println();
                                }

                                writer.println();

                                Debugger.out( "Generated " + groupRatingList.size() + " similarities for item "
+ itemID );
                        }
```

```
                writer.close();

                Debugger.outTimerMIN( "DataFileGenerator", "DataFileGenerator done." );
            }
            catch ( InstantiationException ie )
            {
                System.out.println();
                System.out.println( "usage:" );
                System.out.println( "   java DataFileGenerator [item type]" );
                System.out.println( "   java DataFileGenerator [item type] [rating type]" );
            }
            catch ( Exception e )
            {
                e.printStackTrace();
            }
        }
}
```

```
package com.launch.rm.lc.SimilaritiesEngine;

import com.inet.tds.TdsDriver;
import com.launch.rm.lc.PlaylistGenerator.Constants;
import java.sql.*;
import java.util.*;


/**
 * A database connection. Carries out database operations such as executing
 * SQL queries. There is only one static connection object, which can
 * create multiple statements for executing SQL and return multiple
 * result sets.
 *
 * @author Jeff Boulter
 * @author John Veilleux
 */
public class DBConnection
{
        private final static String DEFAULT_CONN_ID = "DEFAULT";
        private static Driver dbDriver = null;
        private static Hashtable connHash = new Hashtable();
        private static Hashtable connectStrHash = new Hashtable();

        static
        {
                connectStrHash.put( DEFAULT_CONN_ID, "jdbc:inetdae:"
                                + Constants.DB_SERVER
                                + ":"
                                + Constants.DB_PORT
                                + "?sql7=true&database="
                                + Constants.DB_DBNAME
                                + "&user="
                                + Constants.DB_USERNAME
                                + "&password="
                                + Constants.DB_PASSWORD );
        }


        /**
         * Adds a database connection ID and info to the pool.
         *
         * @param       connIDStr       the ID of the new connection
         * @param       connectStr      the connection info
         */
        public final static void addConnection( String connIDStr, String connectStr )
        {
                connectStrHash.put( connIDStr, connectStr );
        }

        /**
         * Initializes the Connection object and adds it to the pool,
         * or does nothing if the object is already initialized,
         * then returns it.
         *
         * @exception   SQLException    if a connection error occurs
         */
        private final static Connection initConnection( String connIDStr ) throws SQLException
        {
                Connection conn;
```

```
            String url;

            conn = (Connection) connHash.get( connIDStr );
            url = (String) connectStrHash.get( connIDStr );

            if ( dbDriver == null )
            {
                dbDriver = new com.inet.tds.TdsDriver();
            }

            if ( dbDriver != null && url != null && ( conn == null || conn.isClosed() ) )
            {
                conn = dbDriver.connect( url, null );

                connHash.put( connIDStr, conn );
            }

            return conn;
        }

        /**
         * Executes an SQL query.
         *
         * @param       sql                 the query to execute
         * @param       printSQL        determines whether or not to print debug info
         *
         * @return                          the result set for the query, or null if
         *                                      an error occurs
         */
        public final static ResultSet executeSQL( String sql, boolean printSQL )
        {
            return executeSQL( DEFAULT_CONN_ID, sql, printSQL );
        }

        /**
         * Executes an SQL query.
         *
         * @param       sql                 the query to execute
         * @param       printSQL        determines whether or not to print debug info
         *
         * @return                          the result set for the query, or null if
         *                                      an error occurs
         */
        public final static ResultSet executeSQL( String connIDStr, String sql, boolean printSQL )
        {
            Connection conn;
            ResultSet rs;
            Statement st;

            conn = null;
            rs = null;
            st = null;

            // if we don't have a query, don't run it--it'll hang
            if ( sql.length() <= 0 )
            {
                System.err.println( new java.util.Date() + " DBConnection.executeSQL: can't run empty
SQL query." );

                return null;
```

```
        }

        if ( printSQL )
        {
            System.out.println( "Running SQL: " + sql );
        }

        try
        {
            conn = initConnection( connIDStr );
            st = conn.createStatement();

            st.execute( sql );

            rs = st.getResultSet();
        }
        catch ( SQLException sqle )
        {
            System.err.println( new java.util.Date() + " Error running SQL: " + sql );
            sqle.printStackTrace();
        }

        return rs;
    }

    /**
     * Executes an SQL update.
     *
     * @param       sql                 the update to execute
     * @param       printSQL        determines whether or not to print debug info
     */
    public final static void executeUpdate( String sql, boolean printSQL )
    {
        executeUpdate( DEFAULT_CONN_ID, sql, printSQL );
    }

    /**
     * Executes an SQL update.
     *
     * @param       sql                 the update to execute
     * @param       printSQL        determines whether or not to print debug info
     */
    public final static void executeUpdate( String connIDStr, String sql, boolean printSQL )
    {
        Connection conn;
        Statement st;

        conn = null;
        st = null;

        // if we don't have a query, don't run it--it'll hang
        if ( sql.length() <= 0 )
        {
            System.err.println( new java.util.Date() + " DBConnection.executeUpdate: can't run
empty SQL query." );

            return;
        }

        if ( printSQL )
```

```
            {
                System.out.println( "Running SQL: " + sql );
            }

            try
            {
                conn = initConnection( connIDStr );
                st = conn.createStatement();

                st.executeUpdate( sql );
            }
            catch ( SQLException sqle )
            {
                System.err.println( new java.util.Date() + " Error running SQL: " + sql );
                sqle.printStackTrace();
            }
        }

        /**
         * Gets a DBPreparedStatement object given an SQL query.
         *
         * @param       sql                 the query to prepare
         *
         * @return                          the prepared statement
         *
         * @exception   SQLException    if a database error occurs
         */
        public final static PreparedStatement prepareStatement( String sql ) throws SQLException
        {
            return prepareStatement( DEFAULT_CONN_ID, sql );
        }

        /**
         * Gets a DBPreparedStatement object given an SQL query.
         *
         * @param       sql                 the query to prepare
         *
         * @return                          the prepared statement
         *
         * @exception   SQLException    if a database error occurs
         */
        public final static PreparedStatement prepareStatement( String connIDStr, String sql ) throws
SQLException
        {
            PreparedStatement ps;
            Connection conn;

            ps = null;
            conn = initConnection( connIDStr );

            if ( conn != null )
            {
                ps = conn.prepareStatement( sql );
            }

            return ps;
        }

        /**
         * Closes a single database connection. It is removed from
```

```
        * the pool of connections.
        *
        * @param        connIDStr       the connection ID
        */
        public final static void closeConnection( String connIDStr )
        {
                Connection conn;

                conn = (Connection) connHash.get( connIDStr );

                try
                {
                        connHash.remove( connIDStr );

                        if ( conn != null )
                        {
                                conn.close();
                        }
                }
                catch ( Exception e )
                {
                        e.printStackTrace();
                }
        }

        /**
        * Closes all database connections in the pool.
        */
        public final static void closeAllConnections()
        {
                Connection conn;
                String connIDStr;

                conn = null;
                connIDStr = null;

                for ( Enumeration enum = connHash.keys(); enum.hasMoreElements(); )
                {
                        try
                        {
                                connIDStr = (String) enum.nextElement();
                                conn = (Connection) connHash.get( connIDStr );

                                connHash.remove( connIDStr );
                                conn.close();
                        }
                        catch ( Exception e )
                        {
                                e.printStackTrace();
                        }
                }
        }
}
```

```
package com.launch.rm.lc.SimilaritiesEngine;

import java.util.Hashtable;
import java.io.*;


/**
 * This class handles all debugging functions, such as debug output,
 * for the SimilaritiesEngine package.
 *
 * @author        John Veilleux
 */
public class Debugger
{
        private static Hashtable timerHash = new Hashtable();
        private static PrintStream outStream = new PrintStream( System.out );

        static
        {
                if ( SimilaritiesConstants.DEBUG && SimilaritiesConstants.LOGFILE )
                {
                        try
                        {
                                outStream = new PrintStream( new FileOutputStream( "SimilaritiesLog.txt" ) );
                        }
                        catch ( Exception e )
                        {
                                System.err.println( "Could not create log file...debug info will be printed to
standard out." );
                        }
                }
        }


        /**
         * Outputs the given message if debug mode is on.
         *
         * @param        message        the message to print
         */
        public final static void out( String message )
        {
                if ( SimilaritiesConstants.DEBUG )
                {
                        outStream.println( "DEBUGGER: " + message );
                }
        }

        /**
         * Outputs the given message with the current timer value in
         * both milliseconds and minutes if debug mode is on.
         *
         * @param        timerKey        the timer ID
         * @param        message        the message to print
         */
        public final static void outTimer( Object timerKey, String message )
        {
                if ( SimilaritiesConstants.DEBUG )
                {
                        if ( timerHash.get( timerKey ) != null )
                        {
```

```
                        outStream.println( "DEBUGGER (" + getTimerMS( timerKey ) + " MS or " +
getTimerMIN( timerKey ) + " MIN): " + message );
                    }
                    else
                    {
                        outStream.println( "DEBUGGER (NO TIMER FOUND): " + message );
                    }
                }
            }

            /**
             * Outputs the given message with the current timer value in
             * milliseconds if debug mode is on.
             *
             * @param       timerKey        the timer ID
             * @param       message         the message to print
             */
            public final static void outTimerMS( Object timerKey, String message )
            {
                if ( SimilaritiesConstants.DEBUG )
                {
                    if ( timerHash.get( timerKey ) != null )
                    {
                        outStream.println( "DEBUGGER (" + getTimerMS( timerKey ) + " MS): " +
message );
                    }
                    else
                    {
                        outStream.println( "DEBUGGER (NO TIMER FOUND): " + message );
                    }
                }
            }

            /**
             * Outputs the given message with the current timer value in
             * minutes if debug mode is on.
             *
             * @param       timerKey        the timer ID
             * @param       message         the message to print
             */
            public final static void outTimerMIN( Object timerKey, String message )
            {
                if ( SimilaritiesConstants.DEBUG )
                {
                    if ( timerHash.get( timerKey ) != null )
                    {
                        outStream.println( "DEBUGGER (" + getTimerMIN( timerKey ) + " MIN):" +
message );
                    }
                    else
                    {
                        outStream.println( "DEBUGGER (NO TIMER FOUND): " + message );
                    }
                }
            }

            /**
             * Resets the timer.
             *
             * @param       timerKey        the timer ID
```

```
    */
    public final static void resetTimer( Object timerKey )
    {
        timerHash.put( timerKey, new Long( System.currentTimeMillis() ) );
    }

    /**
     * Gets the timer's current value in milliseconds.
     *
     * @param       timerKey        the timer ID
     *
     * @return                      the timer's value in milliseconds
     */
    public final static long getTimerMS( Object timerKey )
    {
        Long timerMS;

        timerMS = (Long) timerHash.get( timerKey );

        return System.currentTimeMillis() - timerMS.longValue();
    }

    /**
     * Gets the timer's current value in minutes.
     *
     * @param       timerKey        the timer ID
     *
     * @return                      the timer's value in minutes
     */
    public final static int getTimerMIN( Object timerKey )
    {
        Long timerMS;

        timerMS = (Long) timerHash.get( timerKey );

        return (int) ( ( System.currentTimeMillis() - timerMS.longValue() ) / 60000 );
    }
}
```

```
package com.launch.rm.lc.SimilaritiesEngine;


/**
 * This class calculates the group rating for a single item. The
 * value is calculated by multiplying the total number of ratings
 * by the sum of the average of the ratings with some specified
 * offset.
 *
 * @author Jeff Boulter
 * @author John Veilleux
 */
public class GroupRating
{
        private int itemID;
        private int numRatings = 0;
        private int ratingsSum = 0;
        private double value = 0;
        private double average = 0;
        private boolean stale = true;


        /**
         * Creates a GroupRating object.
         *
         * @param        itemID   the item ID
         */
        public GroupRating( int itemID )
        {
                this.itemID = itemID;
        }

        /**
         * Gets the item ID associated with this group rating.
         *
         * @return       the item ID
         */
        public int getItemID()
        {
                return itemID;
        }

        /**
         * Adds a rating to be used in the calculation of this object's
         * value.
         *
         * @param        r        the rating
         */
        public void add( Rating r )
        {
                numRatings ++;
                ratingsSum += r.value;
                stale = true;
        }

        /**
         * Gets the final value of this object. If the value hasn't
         * been calculated yet, it is calculated and then returned.
         *
         * @return       this object's value
```

```
        */
        public double get()
        {
                if ( stale )
                {
                        calculate();
                }

                return value;
        }

        /**
         * Gets the number of ratings added to this object.
         *
         * @return          the rating count
         */
        public int getNumRatings()
        {
                return numRatings;
        }

        /**
         * Gets a String representation of this object.
         *
         * @return          the String description
         */
        public String toString()
        {
                return String.valueOf( itemID );
        }

        /**
         * Gets a more complete String representation of this object.
         *
         * @return          the String description
         */
        public String toBigString()
        {
                return "itemID: " + itemID + ", # of ratings: " + numRatings + ", sum of ratings: " + ratingsSum +
", average: " + average() + ", score: " + get();
        }

        /**
         * Gets the average value of all of this object's ratings.
         *
         * @return          the rating average
         */
        private double average()
        {
                if ( stale )
                {
                        if ( numRatings <= 0 )
                        {
                                average = 0;
                        }
                        else
                        {
                                average = ( (double)ratingsSum ) / ( (double)numRatings );
                        }
                }
```

```
            return average;
        }

        /**
         * Calculates the value for this object.
         */
        private void calculate()
        {
            value = numRatings * ( average() + SimilaritiesConstants.GR_AVG_OFFSET );
            stale = false;
        }
}
```

```
package com.launch.rm.lc.SimilaritiesEngine;

import java.util.Vector;


/**
 * This class represents a list of OrderedElement objects. They
 * are sorted from highest to lowest value using quicksort. The
 * sorting is done on demand whenever any information contained in
 * this object is accessed.
 *
 * @author Jeff Boulter
 * @author John Veilleux
 */
public class OrderedList
{
        private Vector list;
        private boolean sorted = false;

        /**
         * This inner class represents an element used by OrderedList. It contains
         * two fields that are accessed directly: a value and an associated
         * object. OrderedList sorts these objects by the value field.
         */
        private class OrderedElement
        {
                private double value;
                private Object thing;


                /**
                 * Creates an OrderedElement object.
                 *
                 * @param        value    this object's value
                 * @param        thing    the object associated with the given value
                 */
                private OrderedElement( double value, Object thing )
                {
                        this.value = value;
                        this.thing = thing;
                }
        }


        /**
         * Creates an OrderedList object.
         */
        public OrderedList()
        {
                list = new Vector();
        }

        /**
         * Creates an OrderedList object with an initial size.
         *
         * @param        size       the initial size
         */
        public OrderedList( int size )
        {
                list = new Vector( size );
```

```
}

/**
 * Creates an OrderedList object with an initial size and a
 * capacity increment.
 *
 * @param       size                    the initial size
 * @param       capacityIncrement       the capacity increment
 */
public OrderedList( int size, int capacityIncrement )
{
        list = new Vector( size, capacityIncrement );
}

/**
 * Gets the object at the specified index.
 *
 * @param       int             the index position of the object
 *
 * @return                      the object, or null if no object could be
 *                                      retrieved at the given index
 */
public Object elementAt( int i )
{
        Object obj = null;
        OrderedElement e = null;

        if ( !sorted )
        {
                sort();
        }

        e = (OrderedElement) list.elementAt( i );

        if ( e != null )
        {
                obj = e.thing;
        }

        return obj;
}

/**
 * Gets the value at the specified index.
 *
 * @param       i       the index position of the value
 *
 * @return              the value, or null if no value could be
 *                              retrieved at the given index
 */
public double valueAt( int i )
{
        double value = 0;
        OrderedElement e = null;

        if ( !sorted )
        {
                sort();
        }
```

```
        e = (OrderedElement) list.elementAt( i );

        if ( e != null )
        {
                value = e.value;
        }

        return value;
}

/**
 * Gets the number of elements in the list.
 *
 * @return          the list size
 */
public int size()
{
        return list.size();
}

/**
 * Truncates the list to the specified size. Nothing happens
 * if the list is already equal to or smaller than the given
 * size.
 *
 * @param          size       the maximum size
 */
public void trimToMaximumSize( int size )
{
        if ( !sorted )
        {
                sort();
        }

        if ( list.size() > size )
        {
                list.setSize( size );
        }
}

/**
 * Gets this list as a Vector of the objects associated with
 * each element in this list.
 *
 * @return          the Vector of objects
 */
public Vector asVector()
{
        Vector result = new Vector();

        if ( !sorted )
        {
                sort();
        }

        for ( int i = 0; i < list.size(); i ++ )
        {
                result.addElement( elementAt( i ) );
        }
```

```
        return result;
    }

    /**
     * Gets a String representation of this object.
     *
     * @return      the String description
     */
    public String toString()
    {
        String result = "(";

        if ( !sorted )
        {
            sort();
        }

        for ( int i = 0; i < list.size(); i ++ )
        {
            result += elementAt( i ) + ", ";
        }

        result += ")";

        return result;
    }

    /**
     * Adds a value/object pair to the list.
     *
     * @param       value   the value
     * @param       object  the object
     */
    public void add( double value, Object toStore )
    {
        list.addElement( new OrderedElement( value, toStore ) );

        sorted = false;
    }

    /**
     * Removes an element from the list.
     *
     * @param       index   the index of the element to remove
     */
    public void removeElementAt( int index )
    {
        list.removeElementAt( index );
    }

    /**
     * Sorts this object.
     */
    private void sort()
    {
        sort( list, 0, list.size() - 1 );

        sorted = true;
    }
```

```
/**
 * Performs quick sort on a vector.
 *
 * @param       a               the vector to sort
 * @param       from    the starting index for the sort
 * @param       to              the ending index for the sort
 */
private final static void sort( Vector a, int from, int to )
{
        int i = from;
        int j = to;
        OrderedElement center = null;
        OrderedElement temp = null;

        if ( a == null || a.size() < 2 )
        {
                return;
        }

        center = (OrderedElement) a.elementAt( ( from + to ) / 2 );

        do
        {
                while ( i < to && center.value < ( (OrderedElement) a.elementAt( i ) ).value )
                {
                        i++;
                }

                while ( j > from && center.value > ( (OrderedElement) a.elementAt( j ) ).value )
                {
                        j--;
                }

                if ( i < j )
                {
                        // swap elements
                        temp = (OrderedElement) a.elementAt( i );

                        a.setElementAt( a.elementAt( j ), i );
                        a.setElementAt( temp, j );
                }

                if ( i <= j )
                {
                        i++;
                        j--;
                }

        }
        while( i <= j );

        if ( from < j )
        {
                sort( a, from, j );
        }

        if ( i < to )
        {
                sort( a, i, to );
        }
```

```
        }
}
```

```
package com.launch.rm.lc.SimilaritiesEngine;


/**
 * This class represents a rating. It includes three fields: an item
 * ID, a user ID, and a value. The fields are accessed directly.
 *
 * @author Jeff Boulter
 * @author John Veilleux
 */
public class Rating
{
        public int itemID;
        public int userID;
        public byte value;


        /**
         * Creates a Rating object.
         *
         * @param       itemID          the ID of the item this rating is for
         * @param       userID          the ID of the user who created the rating
         * @param       value           the actual rating value
         */
        public Rating( int itemID, int userID, byte value )
        {
                this.itemID = itemID;
                this.userID = userID;
                this.value = value;
        }

        /**
         * Gets a String representation of this object.
         *
         * @return      the String description
         */
        public String toString()
        {
                return "Rating: [itemID: " + itemID + ", userID: " + userID + ", value: " + value + "]";
        }
}
```

```
package com.launch.rm.lc.SimilaritiesEngine;

import com.launch.utils.PropertiesFileReader;


/**
 * Constants used within the SimilaritiesEngine code. Changing
 * certain parameters can significantly change the amount of memory
 * used. For instance, each rating loaded into the engine uses about
 * 30 bytes of memory, so increasing MAX_RATINGS_IN_ENGINE by 1
 * million ratings could potentially use an extra 30 MB of memory.
 * Each fan under MAX_FANS_PER_ITEM uses about 23 bytes, so
 * MAX_ITEMS_TO_STORE times MAX_FANS_PER_ITEM times 23 bytes gives
 * you the potential maximum amount of memory taken up by those
 * parameters. The ITEM_TO_ARTIST_CACHE_MAX_SIZE entries each use
 * up about 71 bytes of memory. A cache with 15,000 entries will
 * use about 1 MB of memory.
 *
 * @author Jeff Boulter
 * @author John Veilleux
 */
public class SimilaritiesConstants
{
        private final static PropertiesFileReader pfr = new PropertiesFileReader(
"SimilaritiesConstants.properties" );
        private static int maxRatingsInEngine;
        private static String fileNames[] = { "", "", "", "" };
        private static long updateSimilaritiesTimeMS;
        private static short maxItemsToStore[] = { 0, 0, 0, 0 };
        private static int maxSimilarItemsPerItem;
        private static byte fanThreshold;
        private static int maxFansPerItem;
        private static int minFansForRecommendedItem;
        private static int grAvgOffset;
        private static int itemToArtistCacheMaxSize[] = { 0, 0, 0, 0 };
        private static boolean debug;
        private static boolean logfile;

        static
        {
                maxRatingsInEngine = pfr.getIntProperty( "MAX_RATINGS_IN_ENGINE", 30000000 );
                fileNames[ 1 ] = pfr.getProperty( "SONG_RATINGS_FILE", "\\export\\songratings.txt" );
                fileNames[ 2 ] = pfr.getProperty( "ALBUM_RATINGS_FILE", "\\export\\albumratings.txt" );
                fileNames[ 3 ] = pfr.getProperty( "ARTIST_RATINGS_FILE", "\\export\\artistratings.txt" );
                fileNames[ 0 ] = fileNames[ 3 ];
                updateSimilaritiesTimeMS = pfr.getLongProperty( "UPDATE_SIMILARITIES_TIME_MS",
1000 * 60 * 60 * 24 * 14 );
                maxItemsToStore[ 1 ] = pfr.getShortProperty( "MAX_SONGS_TO_STORE", (short)15000 );
                maxItemsToStore[ 2 ] = pfr.getShortProperty( "MAX_ALBUMS_TO_STORE", (short)10000 );
                maxItemsToStore[ 3 ] = pfr.getShortProperty( "MAX_ARTISTS_TO_STORE", (short)3000 );
                maxItemsToStore[ 0 ] = maxItemsToStore[ 3 ];
                maxSimilarItemsPerItem = pfr.getIntProperty( "MAX_SIMILAR_ITEMS_PER_ITEM", 100 );
                fanThreshold = pfr.getByteProperty( "FAN_THRESHOLD", (byte)90 );
                maxFansPerItem = pfr.getIntProperty( "MAX_FANS_PER_ITEM", 300 );
                minFansForRecommendedItem = pfr.getIntProperty(
"MIN_FANS_FOR_RECOMMENDED_ITEM", 4 );
                grAvgOffset = pfr.getIntProperty( "GR_AVG_OFFSET", -70 );
                itemToArtistCacheMaxSize[ 1 ] = pfr.getIntProperty(
"SONG_TO_ARTIST_CACHE_MAX_SIZE", 300000 );
```

```
            itemToArtistCacheMaxSize[ 2 ] = pfr.getIntProperty(
"ALBUM_TO_ARTIST_CACHE_MAX_SIZE", 150000 );
            debug = pfr.getBooleanProperty( "DEBUG", true );
            logfile = pfr.getBooleanProperty( "LOGFILE", false );
        }


        // the maximum number of ratings that the engine can load without
        // running out of memory
        public final static int MAX_RATINGS_IN_ENGINE = maxRatingsInEngine;

        // the file names for the corresponding item type
        // the array is indexed as { 0 = default (artists), 1 = songs, 2 = albums, 3 = artists }
        public final static String FILE_NAMES[] = fileNames;

        // the expiration time for similarities in the database
        public final static long UPDATE_SIMILARITIES_TIME_MS = updateSimilaritiesTimeMS;

        // the maximum number of items with similar items to be stored in the database
        // the array is indexed as { 0 = default (artists), 1 = songs, 2 = albums, 3 = artists }
        public final static short MAX_ITEMS_TO_STORE[] = maxItemsToStore;

        // the maximum number of similar items to retrieve per item
        public final static int MAX_SIMILAR_ITEMS_PER_ITEM = maxSimilarItemsPerItem;

        // the user's minimum rating for an item to be considered a fan
        public final static byte FAN_THRESHOLD = fanThreshold;

        // maximum number of fans to get for an item
        public final static int MAX_FANS_PER_ITEM = maxFansPerItem;

        // the minimum number of ratings an item needs to be considered as a similar item
        public final static int MIN_FANS_FOR_RECOMMENDED_ITEM = minFansForRecommendedItem;

        // used when calculating the average part of a group rating
        public final static int GR_AVG_OFFSET = grAvgOffset;

        // used to determine the maximum size of the cache that maps
        // item ID's to artist ID's
        // the array is indexed as { 0 = default (artists), 1 = songs, 2 = albums, 3 = artists }
        public final static int ITEM_TO_ARTIST_CACHE_MAX_SIZE[] = itemToArtistCacheMaxSize;

        // determines whether or not to print debug output
        public final static boolean DEBUG = debug;

        // in debug mode, determines whether to print debug info to a
        // file or to the screen
        public final static boolean LOGFILE = logfile;
}
```

```
package com.launch.rm.lc.SimilaritiesEngine;

import com.launch.rm.lc.PlaylistGenerator.*;
import java.util.*;
import java.io.*;
import java.sql.*;


/**
 * This class represents the engine which churns out the item
 * similarities. The files from which the ratings are pulled must
 * be grouped by user.
 *
 * @author Jeff Boulter
 * @author John Veilleux
 */
public class SimilaritiesEngine
{
        private byte itemType = 0;
        private Hashtable userRatingsHash = new Hashtable();
        private Hashtable itemToFanIDsHash = new Hashtable();
        private Hashtable itemToArtistCache = null;
        private final static String CACHE_CONN_ID = "CACHE";

        static
        {
                DBConnection.addConnection( CACHE_CONN_ID, "jdbc:inetdae:"
                                + Constants.DB_SERVER
                                + ":"
                                + Constants.DB_PORT
                                + "?sql7=true"
                                + "&database=dbLaunchProd"
                                + "&user="
                                + Constants.DB_USERNAME
                                + "&password="
                                + Constants.DB_PASSWORD );
        }


        /**
         * Creates a SimilaritiesEngine object.
         *
         * @param       itemType        the item type for which similarities will
         *                                      be generated
         *
         * @param       numItems        the number of items that will have
         *                                      similarities generated for them
         */
        public SimilaritiesEngine( byte itemType, int numItems )
        {
                IntHash itemsToExclude;
                LineNumberReader reader;
                String line;
                StringTokenizer st;
                int itemID;
                int userID;
                byte rating;
                Vector userRatings;
                int lastUserID;
                boolean lastUserWasFan;
```

```
int randomStartLine;
int numItemsWithMaxFans;
boolean allFansLoaded;
Vector fanIDsVec;
int numFileRatings[];
int portionToLoad;
int totalRatingsLoaded;

this.itemType = itemType;
itemsToExclude = null;
reader = null;
line = null;
st = null;
itemID = 0;
userID = 0;
rating = 0;
userRatings = null;
lastUserID = -1;
lastUserWasFan = false;
randomStartLine = 0;
numItemsWithMaxFans = 0;
allFansLoaded = false;
fanIDsVec = null;
itemToArtistCache = new Hashtable(
SimilaritiesConstants.ITEM_TO_ARTIST_CACHE_MAX_SIZE[ itemType ] );
numFileRatings = new int[]{ 0 };
portionToLoad = 1;
totalRatingsLoaded = 0;

try
{
        itemsToExclude = getItemsToExclude( itemType );

        Debugger.out( "There were " + itemsToExclude.size() + " items that already had
similarities in the database and don't need to be updated yet." );
        Debugger.out( "Now getting items with the most total ratings..." );
        Debugger.resetTimer( "getItemsWithMostRatings" );

        itemToFanIDsHash = getItemsWithMostRatings( itemType, numItems,
itemsToExclude, numFileRatings );

        Debugger.outTimer( "getItemsWithMostRatings", "Done getting items with the most
total ratings. # of items: " + itemToFanIDsHash.size() );

        portionToLoad = ( numFileRatings[ 0 ] /
SimilaritiesConstants.MAX_RATINGS_IN_ENGINE ) + 1;
        randomStartLine = (int) Util.random( numFileRatings[ 0 ] ) + 1;
        reader = new LineNumberReader( new FileReader(
SimilaritiesConstants.FILE_NAMES[ itemType ] ) );

        Debugger.out( "Engine will load no more than 1/" + portionToLoad + " of " +
numFileRatings[ 0 ] + " total ratings in file." );
        Debugger.out( "Starting to read ratings file up through random line " + randomStartLine
);

        for ( int i = 1; i <= randomStartLine; i ++ )
        {
                line = reader.readLine();
        }
```

```
                    Debugger.out( "Done reading file up through random line " + randomStartLine );
                    Debugger.out( "Now queuing up file to first line of next user..." );

                    line = readUpToNextUser( line, reader );
                    randomStartLine = reader.getLineNumber();

                    if ( line == null )
                    {
                            reader = new LineNumberReader( new FileReader(
SimilaritiesConstants.FILE_NAMES[ itemType ] ) );
                            line = reader.readLine();
                            randomStartLine = reader.getLineNumber();
                    }

                    Debugger.out( "Done queuing up file to first line of next user." );
                    Debugger.out( "Now loading ratings into engine..." );
                    Debugger.resetTimer( toString() );

                    do
                    {
                            if ( reader.getLineNumber() % portionToLoad == 0 )
                            {
                                    st = new StringTokenizer( line, "," );
                                    itemID = Integer.parseInt( st.nextToken() );
                                    userID = Integer.parseInt( st.nextToken() );
                                    rating = Byte.parseByte( st.nextToken() );

                                    if ( userID != lastUserID )
                                    {
                                            if ( lastUserWasFan )
                                            {
                                                    lastUserWasFan = false;

                                                    userRatingsHash.put( new Integer( lastUserID ),
userRatings );
                                                    totalRatingsLoaded += userRatings.size();
                                            }

                                            lastUserID = userID;
                                            allFansLoaded = numItemsWithMaxFans == numItems;
                                            userRatings = new Vector();
                                    }

                                    userRatings.addElement( new Rating( itemID, userID, rating ) );

                                    if ( rating >= SimilaritiesConstants.FAN_THRESHOLD )
                                    {
                                            fanIDsVec = (Vector) itemToFanIDsHash.get( new Integer(
itemID ) );

                                            if ( fanIDsVec != null && fanIDsVec.size() <
SimilaritiesConstants.MAX_FANS_PER_ITEM )
                                            {
                                                    lastUserWasFan = true;

                                                    fanIDsVec.addElement( new Integer( userID ) );

                                                    if ( fanIDsVec.size() ==
SimilaritiesConstants.MAX_FANS_PER_ITEM )
                                                    {
```

```
                                                numItemsWithMaxFans ++;
                                        }
                                }
                        }
                }

                line = reader.readLine();

                if ( line == null )
                {
                        Debugger.out( "Read past end of " +
SimilaritiesConstants.FILE_NAMES[ itemType ] );

                        reader.close();

                        reader = new LineNumberReader( new FileReader(
SimilaritiesConstants.FILE_NAMES[ itemType ] ) );
                        line = reader.readLine();
                }
        }
        while ( !allFansLoaded && reader.getLineNumber() != randomStartLine );

        reader.close();

        if ( lastUserWasFan )
        {
                userRatingsHash.put( new Integer( userID ), userRatings );
                totalRatingsLoaded += userRatings.size();
        }

        Debugger.outTimer( toString(), "Done loading " + totalRatingsLoaded + " ratings into
engine." );
        Debugger.out( numItemsWithMaxFans + " out of " + itemToFanIDsHash.size() + "
items had maximum of " + SimilaritiesConstants.MAX_FANS_PER_ITEM + " fans." );
    }
    catch ( Exception e )
    {
        e.printStackTrace();
    }
}

/**
 * Gets a sorted list of items similar to the given item. The
 * specified item ID must have been one of the candidates to
 * have similarities generated for it.
 *
 * @param       itemID          the ID of the item to get similar items for
 *
 * @return                      the list of similar items, or an empty
 *                                      list if the item ID wasn't included in the
 *                                      similarities calculations
 */
public OrderedList getSimilar( int itemID )
{
        OrderedList result;
        Consensus c;
        Vector fanIDs;
        Vector userRatings;

        result = new OrderedList();
```

```
                c = new Consensus( itemID );
                fanIDs = (Vector) itemToFanIDsHash.get( new Integer( itemID ) );
                userRatings = null;

                if ( fanIDs != null )
                {
                        for ( int i = 0; i < fanIDs.size(); i ++ )
                        {
                                userRatings = (Vector) userRatingsHash.get( fanIDs.elementAt( i ) );

                                c.addRatings( userRatings );
                        }

                        result = c.poll();

                        if ( itemType == Constants.ITEM_TYPE_SONG )
                        {
                                removeItemsWithSameArtist( itemID, result, "exec sp_lcGetSongDetail_xsxx
", itemToArtistCache, SimilaritiesConstants.ITEM_TO_ARTIST_CACHE_MAX_SIZE[ itemType ] );
                        }
                        else if ( itemType == Constants.ITEM_TYPE_ALBUM )
                        {
                                removeItemsWithSameArtist( itemID, result, "exec
sp_lcGetAlbumDetail_xsxx ", itemToArtistCache,
SimilaritiesConstants.ITEM_TO_ARTIST_CACHE_MAX_SIZE[ itemType ] );
                        }
                }
                else
                {
                        Debugger.out( "The item with ID " + itemID + " was not one of the items that had
similarities calculated for it." );
                }

                return result;
        }

        /**
         * Gets a sorted list of items similar to the given item. The
         * specified item ID must have been one of the candidates to
         * have similarities generated for it.
         *
         * @param       itemID          the ID of the item to get similar items for
         * @param       maxItems        the maximum number of similar items to
         *                                      retrieve
         *
         * @return                              the list of similar items, or an empty
         *                                      list if the item ID wasn't included in the
         *                                      similarities calculations
         */
        public OrderedList getSimilar( int itemID, int maxItems )
        {
                OrderedList result;

                result = getSimilar( itemID );

                result.trimToMaximumSize( maxItems );

                return result;
        }
```

```
	/**
	 * Gets all item ID's that need to have their similarities
	 * generated.
	 *
	 * @return		the item ID's
	 */
	public Vector getItemIDs()
	{
		Vector idVec;

		idVec = new Vector( itemToFanIDsHash.size() );

		for ( Enumeration e = itemToFanIDsHash.keys(); e.hasMoreElements(); )
		{
			idVec.addElement( e.nextElement() );
		}

		return idVec;
	}

	/**
	 * Gets an inthash of item ID's to exclude from similarities
	 * generation.
	 *
	 * @param		type	the item type
	 *
	 * @return				the item ID's to exclude
	 */
	private final static IntHash getItemsToExclude( byte type )
	{
		IntHash toExclude;
		ResultSet rs;
		Timestamp lastUpdatedTime;

		toExclude = new IntHash();
		rs = null;
		lastUpdatedTime = null;

		try
		{
			rs = DBConnection.executeSQL( "exec usp_a10xSimilar_GetAllSimilarItems_xsxx " +
type, false );

			while ( rs.next() )
			{
				lastUpdatedTime = rs.getTimestamp( "dateCreated" );

				if ( System.currentTimeMillis() - lastUpdatedTime.getTime() <
SimilaritiesConstants.UPDATE_SIMILARITIES_TIME_MS )
				{
					toExclude.increment( rs.getInt( "itemID" ) );
				}
			}
		}
		catch ( Exception e )
		{
			e.printStackTrace();
		}

		return toExclude;
```

```
	}

	/**
	 * Gets a hashtable with item ID's as the keys and an empty
	 * inthash for each item. There will only be up to specified
	 * maximum number of item ID's in the hashtable, and they will
	 * be chosen from most to least total ratings.
	 *
	 * @param	type			the item type
	 * @param	maxItems		the maximum number of items to return
	 *					in the hashtable
	 * @param	itemsToExclude	a group of item ID's to definitely
	 *					exclude from the returned hashtable
	 * @param	numLines		a one-element array for storing the
	 *					number of lines in the ratings file
	 *
	 * @return				the hashtable of item ID's each with
	 *					an associated inthash
	 */
	private final static Hashtable getItemsWithMostRatings( byte type, int maxItems, IntHash
itemsToExclude, int numLines[] )
	{
		Hashtable resultHash;
		LineNumberReader reader;
		StringTokenizer st;
		int itemID;
		IntHash numRatingsHash;
		OrderedList mostRatingsItemIDList;
		int resultSize;

		resultHash = new Hashtable( maxItems );
		reader = null;
		st = null;
		itemID = 0;
		numRatingsHash = new IntHash();
		mostRatingsItemIDList = new OrderedList();
		resultSize = 0;

		try
		{
			reader = new LineNumberReader( new FileReader(
SimilaritiesConstants.FILE_NAMES[ type ] ) );

			for ( String line = reader.readLine(); line != null; line = reader.readLine() )
			{
				st = new StringTokenizer( line, "," );
				itemID = Integer.parseInt( st.nextToken() );

				if ( itemsToExclude.get( itemID ) == 0 )
				{
					numRatingsHash.increment( itemID );
				}
			}

			numLines[ 0 ] = reader.getLineNumber();

			for ( Enumeration e = numRatingsHash.keys(); e.hasMoreElements(); )
			{
				itemID = ( (Integer) e.nextElement() ).intValue();
```

```
                        mostRatingsItemIDList.add( (double) numRatingsHash.get( itemID ), new
Integer( itemID ) );
                }

                resultSize = Math.min( mostRatingsItemIDList.size(), maxItems );

                for ( int i = 0; i < resultSize; i ++ )
                {
                        resultHash.put( mostRatingsItemIDList.elementAt( i ), new Vector() );
                }
        }
        catch ( Exception e )
        {
                e.printStackTrace();
        }

        return resultHash;
}

/**
 * Removes similar items from the given list that have the same
 * artist as the given item.
 *
 * @param       itemID              the ID of the item whose artist should not
 *                                      be the same as any artists for the items
 *                                      in the given list of similar items
 * @param       simList             the list of items similar to the given item
 * @param       sql                 the sql needed for retrieving the artist ID
 * @param       cache               the cache with item ID's mapped to artist ID's
 * @param       maxCacheSize    the maximum size of the given cache
 */
private final static void removeItemsWithSameArtist( int itemID, OrderedList simList, String sql,
Hashtable cache, int maxCacheSize )
{
        ResultSet rs;
        Integer itemIDInt;
        Integer artistID;
        Integer otherItemID;
        Integer otherArtistID;

        rs = null;
        itemIDInt = new Integer( itemID );
        artistID = (Integer) cache.get( itemIDInt );
        otherItemID = null;
        otherArtistID = null;

        try
        {
                if ( artistID == null )
                {
                        rs = DBConnection.executeSQL( CACHE_CONN_ID, sql + itemID, false );

                        if ( rs.next() )
                        {
                                artistID = new Integer( rs.getInt( "artistID" ) );

                                if ( cache.size() < maxCacheSize )
                                {
                                        cache.put( itemIDInt, artistID );
                                }
```

```
                        }
                        else
                        {
                                artistID = new Integer( -1 );
                        }
                }

                for ( int i = simList.size() - 1; i >= 0; i -- )
                {
                        otherItemID = new Integer( ( (GroupRating) simList.elementAt( i )
).getItemID() );
                        otherArtistID = (Integer) cache.get( otherItemID );

                        if ( otherArtistID == null )
                        {
                                rs = DBConnection.executeSQL( CACHE_CONN_ID, sql +
otherItemID, false );

                                if ( rs.next() )
                                {
                                        otherArtistID = new Integer( rs.getInt( "artistID" ) );

                                        if ( cache.size() < maxCacheSize )
                                        {
                                                cache.put( otherItemID, otherArtistID );
                                        }
                                }
                                else
                                {
                                        otherArtistID = new Integer( artistID.intValue() - 1 );
                                }
                        }

                        if ( artistID.intValue() == otherArtistID.intValue() )
                        {
                                simList.removeElementAt( i );
                        }
                }
        }
        catch ( Exception e )
        {
                e.printStackTrace();
        }
}

/**
 * Reads through lines of a ratings file starting on the line
 * after the given line and returns the first line that has a
 * different user ID than the user ID in the given line.
 *
 * @param       line    the starting line
 * @param       reader  the object reading the ratings file
 *
 * @return                      the first line with a different user, or null
 *                              if the end of the file is reached
 */
private final static String readUpToNextUser( String line, LineNumberReader reader )
{
        StringTokenizer st;
        int firstUserID;
```

```
            int userID;

            st = null;
            firstUserID = 0;
            userID = 0;

            try
            {
                st = new StringTokenizer( line, "," );

                st.nextToken();

                userID = Integer.parseInt( st.nextToken() );
                firstUserID = userID;

                while ( userID == firstUserID )
                {
                    line = reader.readLine();

                    if ( line != null )
                    {
                        st = new StringTokenizer( line, "," );

                        st.nextToken();

                        userID = Integer.parseInt( st.nextToken() );
                    }
                    else
                    {
                        userID = firstUserID - 1;
                    }
                }
            }
            catch ( Exception e )
            {
                e.printStackTrace();
            }

            return line;
        }

        /**
         * Gets a String representation of this object.
         *
         * @return      the String description
         */
        public String toString()
        {
            return "Item Type: " + itemType;
        }
}
```

```
package com.launch.rm.lc.SimilaritiesEngine;

import com.launch.rm.lc.PlaylistGenerator.*;
import java.util.Vector;


/**
 * This class writes similarity data to the database. It takes the
 * item type from the command line.
 *
 * @author John Veilleux
 */
public class SimilaritiesGenerator
{
	/**
	 * The main method.
	 *
	 * @param	args	command line arguments
	 */
	public static void main( String args[] )
	{
		Integer itemID;
		Byte itemType;
		SimilaritiesEngine engine;
		Vector itemIDVec;
		Vector similarIDVec;
		String sql;

		itemID = null;
		itemType = null;
		engine = null;
		itemIDVec = new Vector();
		similarIDVec = null;
		sql = null;

		try
		{
			if ( args.length == 1 )
			{
				itemType = new Byte( args[ 0 ] );
			}
			else
			{
				throw new InstantiationException();
			}

			if ( itemType.byteValue() < Constants.ITEM_TYPE_SONG || itemType.byteValue() >
Constants.ITEM_TYPE_ARTIST )
			{
				throw new Exception( "Item type must be " + Constants.ITEM_TYPE_SONG
+ ", " + Constants.ITEM_TYPE_ALBUM + ", or " + Constants.ITEM_TYPE_ARTIST + "." );
			}

			Debugger.out( "Similarities Generator started." );
			Debugger.resetTimer( "SimilaritiesGenerator" );

			engine = new SimilaritiesEngine( itemType.byteValue(),
SimilaritiesConstants.MAX_ITEMS_TO_STORE[ itemType.intValue() ] );
			itemIDVec = engine.getItemIDs();
```

```
                        for ( int i = 0; i < itemIDVec.size(); i ++ )
                        {
                                itemID = (Integer)itemIDVec.elementAt( i );
                                similarIDVec = engine.getSimilar( itemID.intValue(),
SimilaritiesConstants.MAX_SIMILAR_ITEMS_PER_ITEM ).asVector();
                                sql = "usp_a10xSimilar_SetSimilarItems_ixxd " + itemID + ", " + itemType +
", " + similarIDVec.size() + ", '" + Util.GetVectorAsSpaceDelimitedList( similarIDVec ) + "'";

                                DBConnection.executeUpdate( sql, false );
                                Debugger.out( "Generated " + similarIDVec.size() + " similarities for item " +
itemID );
                        }

                        Debugger.outTimerMIN( "SimilaritiesGenerator", "Similarities Generator done." );
                }
                catch ( InstantiationException ie )
                {
                        System.out.println();
                        System.out.println( "usage:" );
                        System.out.println( "   java SimilaritiesGenerator [item type]" );
                }
                catch ( Exception e )
                {
                        e.printStackTrace();
                }
        }
}
```

CLAIMS

What is claimed is:

1. A method (900) for providing transmission of a data stream according to preferences of a community, the steps comprising:

providing a first community (910), said first community expressing preferences regarding content in first data streams;

determining characteristics of said preferences with regard to said first data streams to provide determined characteristics (920);

biasing an individual data stream according to said determined characteristics so that said individual data stream is biased according to said expressed preferences (930); and

transmitting said individual data stream (940); whereby

said individual data stream has more content that said community likes and less content that said community dislikes.

2. A method for providing transmission of a data stream according to preferences of a community as set forth in Claim 1, wherein said step of biasing an individual data stream further comprises:

biasing an individual data stream according to said determined characteristics so that said individual data stream is biased for positive preferences of said community and biased against negative preferences of said community.

3. The method for providing transmission of a data stream according preferences of a community as set forth in Claim 1, wherein the step of providing a community further comprises:

providing a second community, said second community expressing preferences regarding content in second data streams;

evaluating said preferences of said second community; and

determining said first community from said second community with members of said first community having a preference in common; whereby

said first community arises from a larger second community, said first community determined by having a preference in common.

4. The method for providing transmission of a data stream according preferences of a community as set forth in Claim 1, wherein the step of biasing an individual data stream further comprises:

biasing said individual data stream so that it complies with the DMCA.

5. The method for providing transmission of a data stream according preferences of a community as set forth in Claim 1, wherein the step of transmitting said individual data stream further comprises:

transmitting said individual data stream on a voluntary or selectable basis thereby allowing an individual to receive said individual data stream on a voluntary or selectable basis.

6. A method (900) for providing transmission of a data stream according to preferences of a community,

the steps comprising:

providing a first community (920), said first community expressing first preferences regarding content in first data streams;

providing a second community (910), said second community second expressing preferences regarding content in second data streams;

evaluating said second preferences of said second community; and

determining said first community from said second community with members of said first community having a first preference in common, so that said first community arises from a larger second community, said first community determined by having a first preference in common;

determining characteristics of said first preferences with regard to said first data streams to provide determined characteristics (930);

biasing an individual data stream according to said determined characteristics so that said individual data stream is biased according to said determined characteristics so that said individual data stream is biased for positive first preferences of said first community and biased against negative first preferences of said first community; and

transmitting said individual data stream (940) on a voluntary or selectable basis thereby allowing an individual to receive said individual data stream on a voluntary or selectable basis; whereby

said individual data stream has more content that said community likes and less content that said community dislikes.

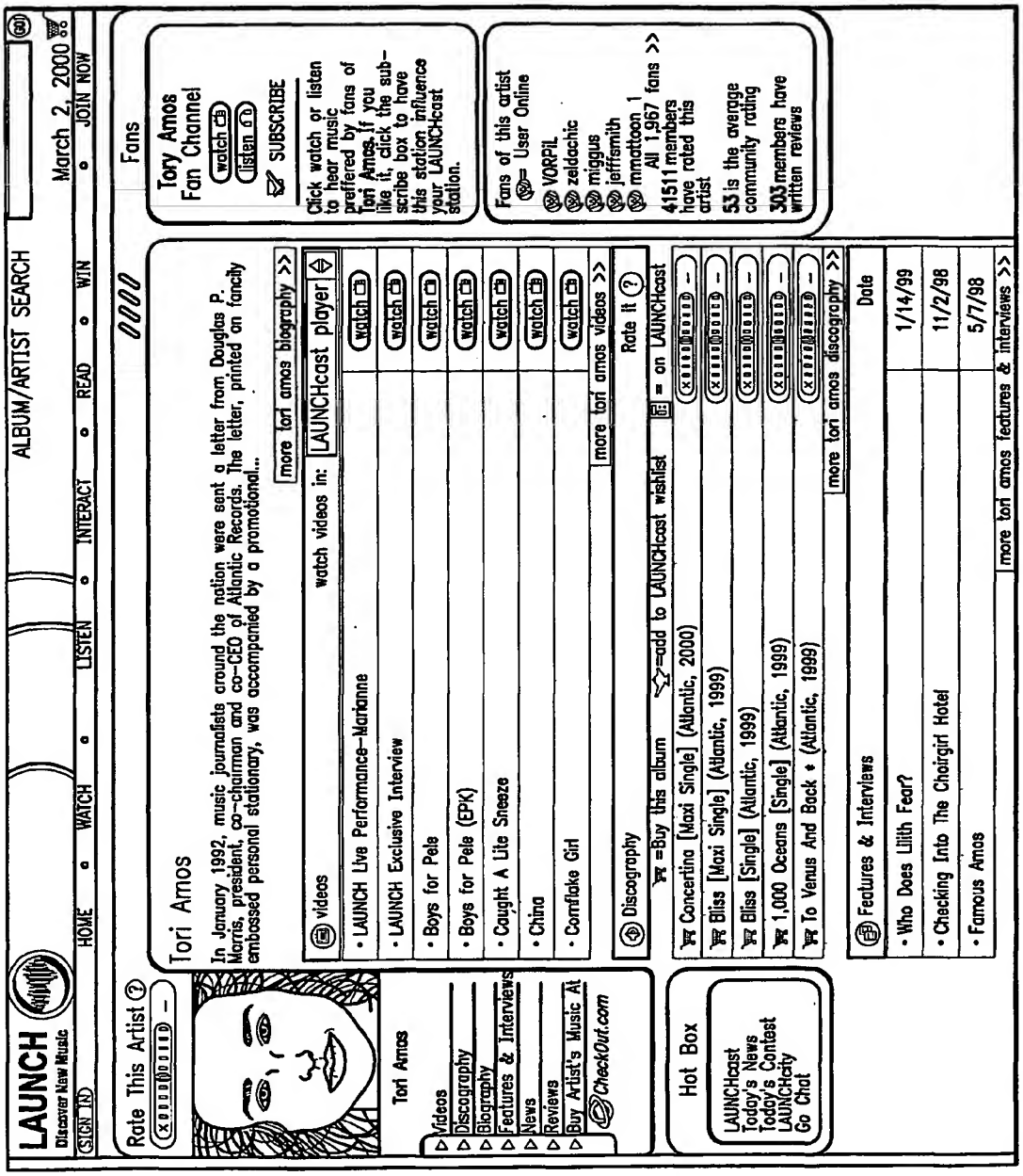


FIG. 1

BEST AVAILABLE COPY

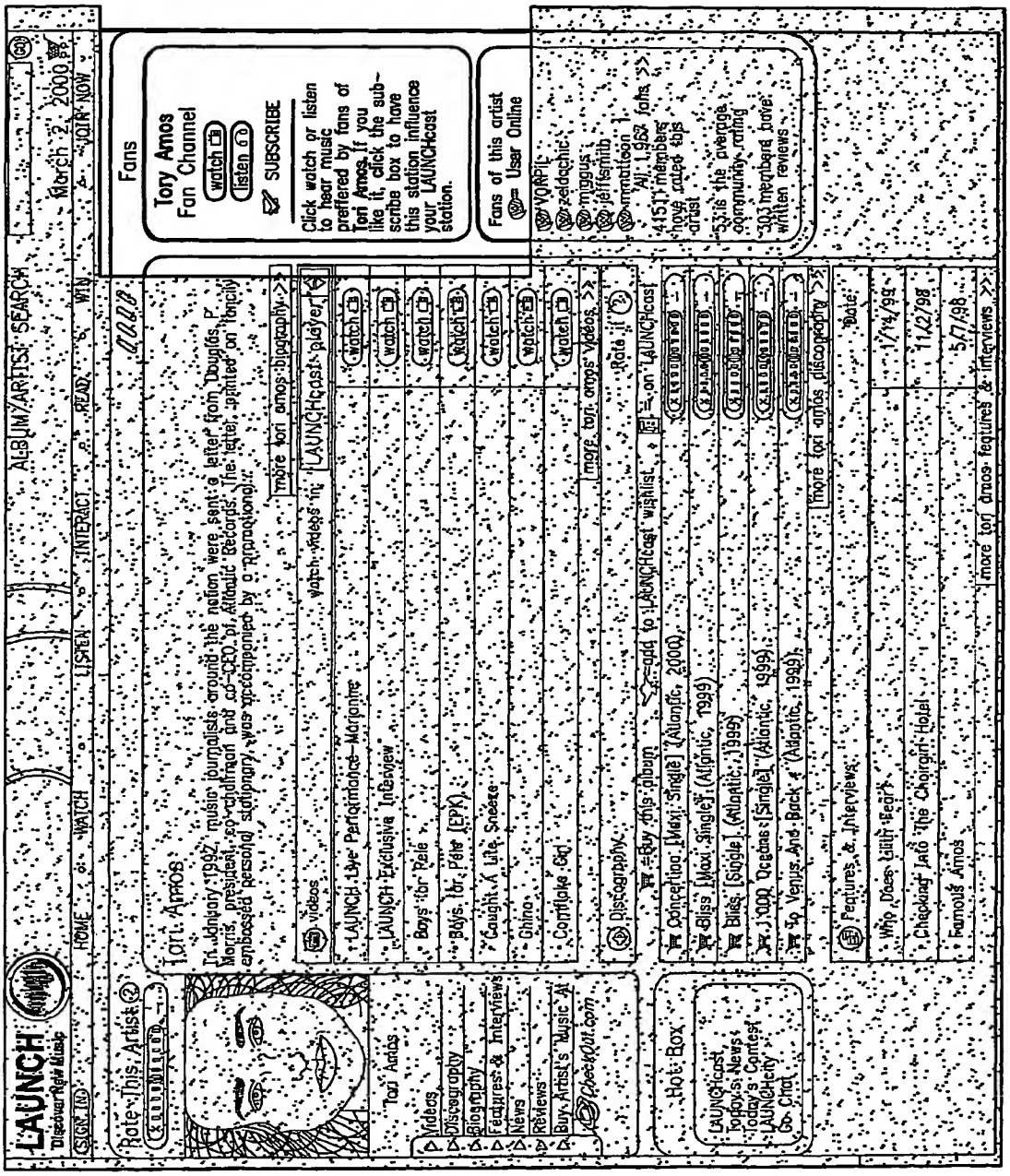


FIG. 2

BEST AVAILABLE COPY

Fans
Tory Amos
Fan Channel
watch
listen
SUBSCRIBE
Click watch or listen to hear music preffered by fans of Tori Amos. If you like it, click the sub-scribe box to have this station influence your LAUNCHcast station.
Fans of this artist
= User Online

FIG. 3

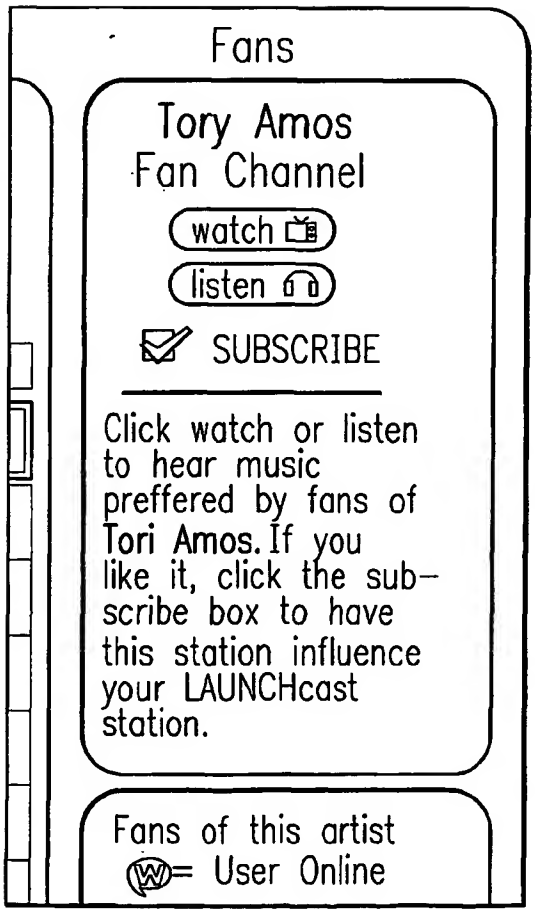

Fans
Tory Amos
Fan Channel
watch
listen
SUBSCRIBE
Click watch or listen to hear music preffered by fans of Tori Amos. If you like it, click the sub-scribe box to have this station influence your LAUNCHcast station.
Fans of this artist
= User Online

FIG. 4

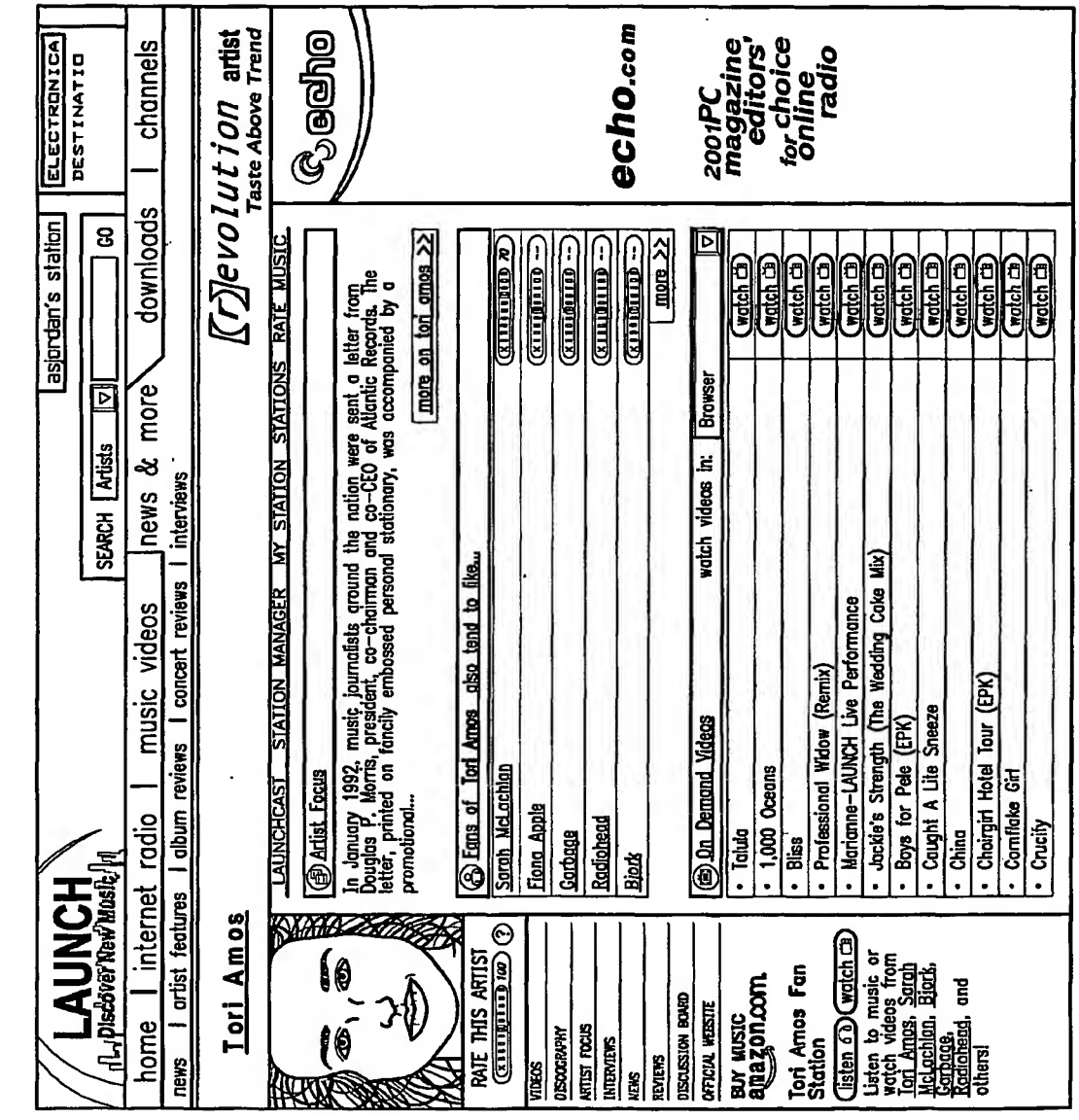


FIG. 5

BEST AVAILABLE COPY

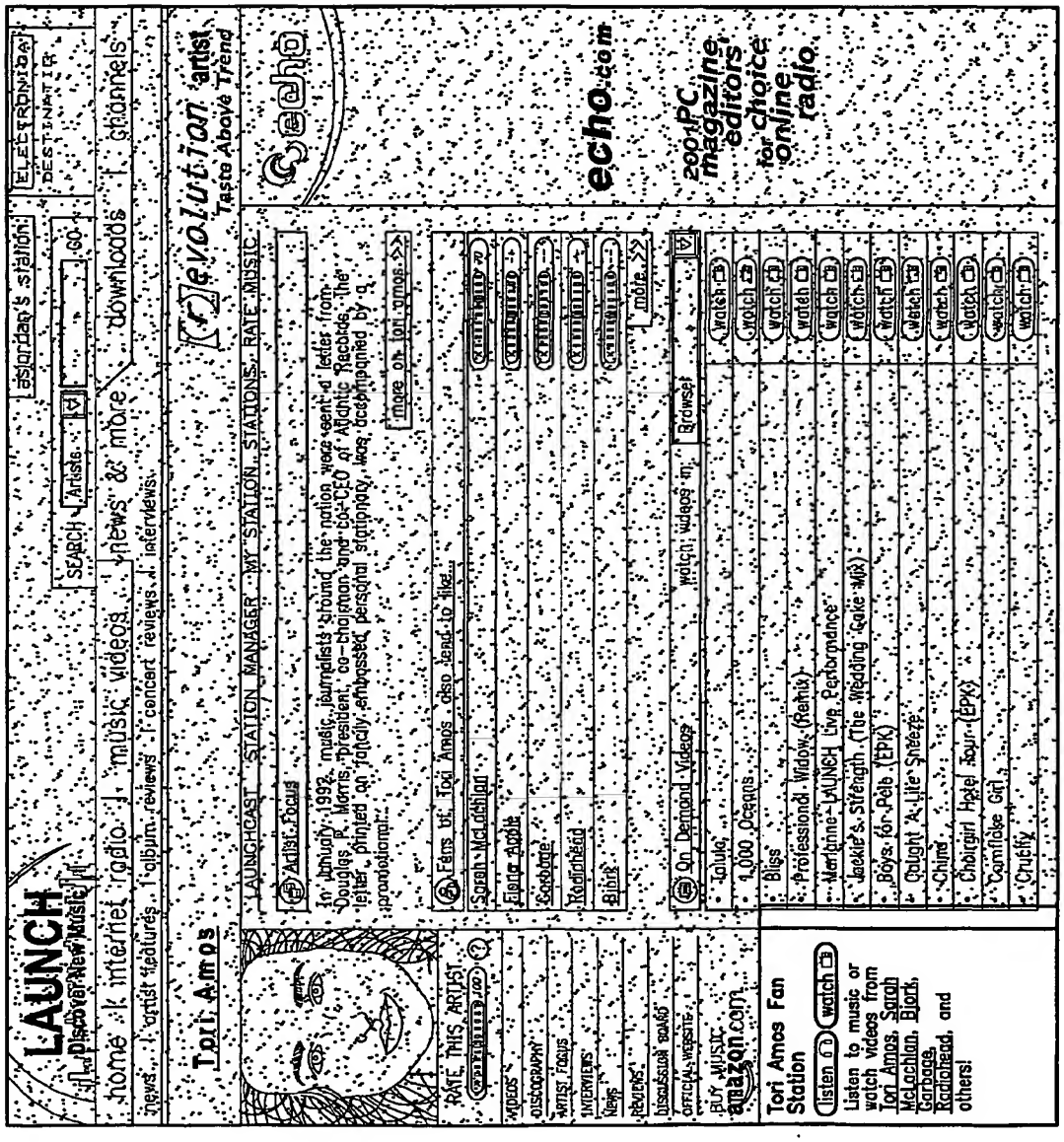


FIG. 6

BEST AVAILABLE COPY

Tori Amos Fan Station
listen
watch
Listen to music or watch videos from Tori Amos, Sarah McLachlan, Bjork, Garbage, Radiohead, and others!

FIG. 7

Tori Amos Fan Station
listen
watch
Listen to music or watch videos from Tori Amos, Sarah McLachlan, Bjork, Garbage, Radiohead, and others!

FIG. 8

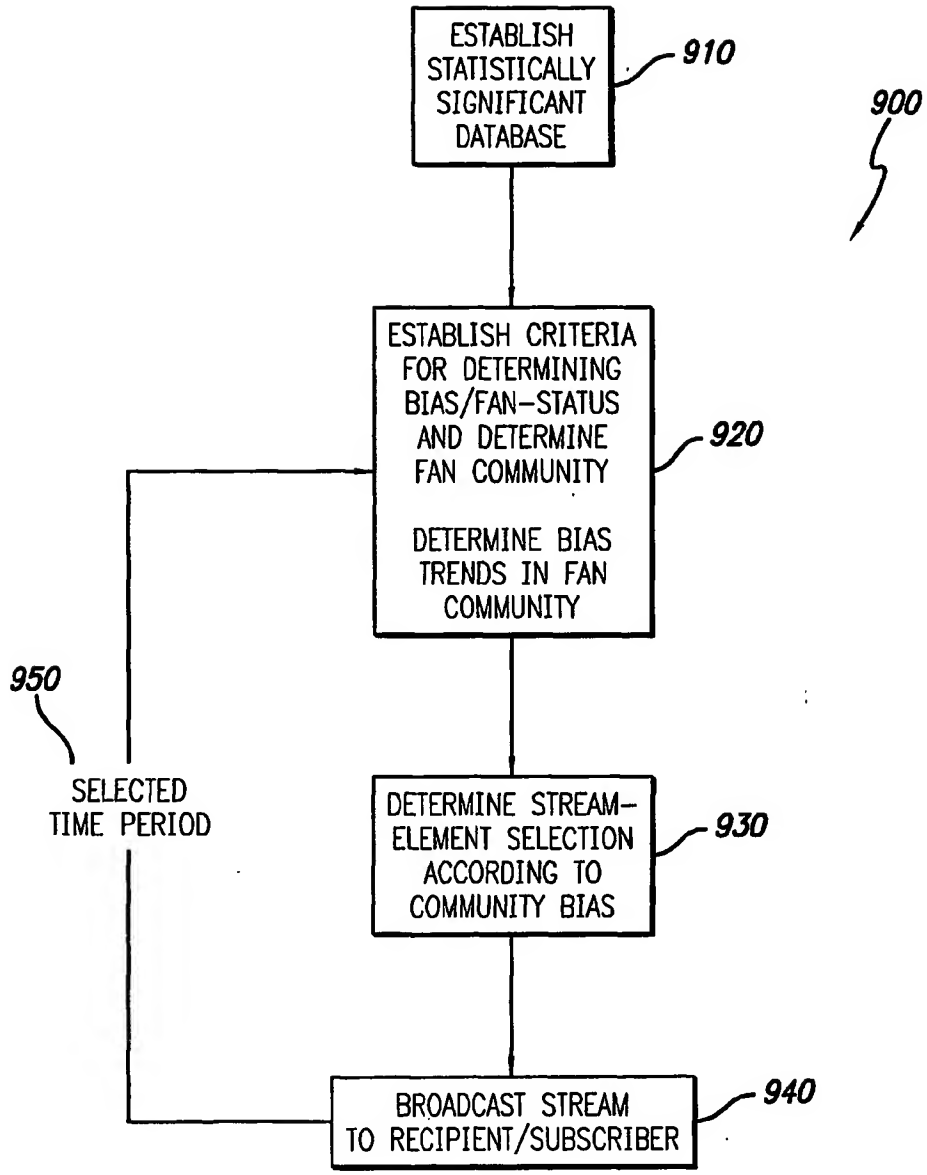

ESTABLISH STATISTICALLY SIGNIFICANT DATABASE
910
900
ESTABLISH CRITERIA FOR DETERMINING BIAS/FAN-STATUS AND DETERMINE FAN COMMUNITY
DETERMINE BIAS TRENDS IN FAN COMMUNITY
920
950
SELECTED TIME PERIOD
DETERMINE STREAM-ELEMENT SELECTION ACCORDING TO COMMUNITY BIAS
930
BROADCAST STREAM TO RECIPIENT/SUBSCRIBER
940

FIG. 9

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US01/21829

A. CLASSIFICATION OF SUBJECT MATTER
IPC(7) : G06F 17/30,15/16;H04B 17/10
US CL : 707/5,10,104.1;725/34,116;704/260;709/217,219
According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
U.S. : 707/5,10,104.1;725/34,116;704/260;709/217,219;345/721

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
WEST

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	US 5,404,505 A (LEVINSON) 04 April 1995, Abstract,Column 18, line 1-62,Column 19, line 27-55;Column 21, line 14-68;Column 22, line 11-25.	1-6
X	US 5,758,257 A (HERTZ et al.) 26 May 1998, Abstract, Figures 1-6;Column 1, line 23 - Column 7, line 12;Column 24, line 57 - Column 27, line 65;Column 29, line 31 - Column 30, line 64; Column 34, line 13 - Column 40, line 20; Column 47, line 21 - Column 48, line 22;Column 52, line 16 - Column 53, line 3; Column 54, line 7-16; Column 57, line 24-64.	1-6
X	US 5,842,010 A (JAIN et al.) 24 November 1998, Abstract;Column 2, line 35 - Column 6, line 2; Column 6, line 49 - Column 12, line 52; Column 13, line 41 - Column 16, line 33.	1-6
X	US 6,020,883 A (HERZ et al..)01 February 2000, Abstract, , Figures 1-6;Column 1, line 23 - Column 7, line 12;Column 24, line 57 - Column 27, line 65;Column 29, line 31 - Column 30, line 64; Column 34, line 13 - Column 40, line 20; Column 47, line 21 - Column 48, line 22;Column 52, line 16 - Column 53, line 3; Column 54, line 7-16; Column 57, line 24-64.	1-6

☐ Further documents are listed in the continuation of Box C. ☐ See patent family annex.

* Special categories of cited documents:

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier application or patent published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
09 September 2001 (09.09.2001)	11 OCT 2001
Name and mailing address of the ISA/US Commissioner of Patents and Trademarks Box PCT Washington, D.C. 20231 Facsimile No. (703)305-3230	Authorized officer Paul R Lintz Telephone No. 305-9600

Form PCT/ISA/210 (second sheet) (July 1998)

(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
17 January 2002 (17.01.2002)

PCT

(10) International Publication Number
WO 02/05140 A1

(51) International Patent Classification[7]: G06F 17/30, 15/16, H04B 17/10

(21) International Application Number: PCT/US01/21829

(22) International Filing Date: 10 July 2001 (10.07.2001)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
60/217,594 11 July 2000 (11.07.2000) US
09/709,234 9 November 2000 (09.11.2000) US

(71) Applicant *(for all designated States except US)*: **LAUNCH MEDIA, INC.** [US/US]; Attn: Legal Department, 2700 Pennsylvania Ave., Santa Monica, CA 90404 (US).

(72) Inventors; and

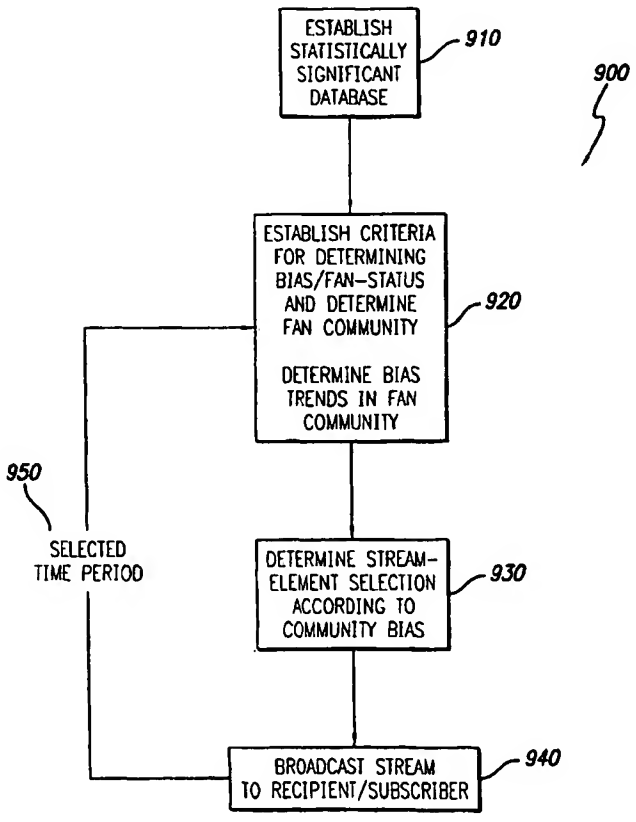
(75) Inventors/Applicants *(for US only)*: **BOULTER, Jeffrey** [US/US]; LAUNCH Media, Inc., Attn: Legal Dept., 2700 Pennsylvania Ave., Santa Monica, CA 90404 (US). **BEAUPRE, Todd** [US/US]; LAUNCH Media, Inc., Attn: Legal Dept., 2700 Pennsylvania Ave., Santa Monica, CA 90404 (US). **VEILLEUX, John-Paul** [US/US]; LAUNCH Media, Inc., Attn: Legal Dept., 2700 Pennsylvania Ave., Santa Monica, CA 90404 (US).

(74) Agents: **JORDAN, Andrew** et al.; Cislo & Thomas LLP, 233 Wilshire Blvd., Ste. 900, Santa Monica, CA 90401-1211 (US).

(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW,

[Continued on next page]

(54) Title: ONLINE PLAYBACK SYSTEM WITH COMMUNITY BIAS



(57) Abstract: A method (900) for entertaining individuals according to a community having similar tastes. Information derived from user accounts (910) form the basis of a community (920) and collateral preferences allow other subscribing individuals to enjoy the benefit of wider-ranging tastes according to the preferences expressed by the other members of the community. Additionally, assuming that individuals sharing one preference in common may be likely to share others, the present method allows those who choose to listen to the "fan station" the ability to enjoy similar music or other data streams according to preferences expressed by the fan community as a whole (930).

WO 02/05140 A1

MX, MZ, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, US, UZ, VN, YU, ZA, ZW.

(84) **Designated States** *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published:
- *with international search report*
- *with amended claims and statement*

Date of publication of the amended claims and statement:
28 February 2002

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

AMENDED CLAIMS

[received by the International Bureau on 11 December 2001 (11.12.01);
original claims 1, 3 and 6 amended; remaining claims unchanged (2 pages)]

1. A method (900) for providing transmission of a data stream according to preferences of a community, the steps comprising:

receiving expressed preferences of a first community having a plurality of members (910), said expressed preferences regarding content in first data streams arising from a first music-related database including songs and/or music videos;

determining characteristics of said preferences with regard to said first data streams to provide determined characteristics (920);

biasing an individual data stream also arising from said first music-related database according to said determined characteristics so that said individual data stream is biased according to said expressed preferences (930); and

transmitting said individual data stream (940); whereby

said individual data stream has more content that said first community likes and less content that said first community dislikes.

2. A method for providing transmission of a data stream according to preferences of a community as set forth in Claim 1, wherein said step of biasing an individual data stream further comprises:

biasing an individual data stream according to said determined characteristics so that said individual data stream is biased for positive preferences of said first community and biased against negative preferences of said first community.

3. The method for providing transmission of a data stream according preferences of a community as set forth in Claim 1, wherein the step of providing a community further comprises:

receiving expressed preferences of a second community having a plurality of members, said expressed preferences regarding content in second data streams arising from said first music-related database;

evaluating said preferences of said second community; and

determining said first community from said second community with members of said first community having a preference in common; whereby

said first community arises from a larger second community, said first community determined by having a preference in common.

4. The method for providing transmission of a data stream according preferences of a community as set forth in Claim 1, wherein the step of biasing an individual data stream further comprises:

biasing said individual data stream so that it complies with the DMCA.

5. The method for providing transmission of a data stream according preferences of a community as set forth in Claim 1, wherein the step of transmitting said individual data stream further comprises:

transmitting said individual data stream on a voluntary or selectable basis thereby allowing an

individual to receive said individual data stream on a voluntary or selectable basis.

6. A method (900) for providing transmission of a data stream according to preferences of a community, the steps comprising:

receiving expressed preferences of a first community having a plurality of members (920), said expressed preferences regarding content in first data streams arising from a first music-related database including songs and/or music videos;

receiving expressed preferences of a second community having a plurality of members (910), said expressed preferences regarding content in second data streams arising from said first music-related database;

evaluating said second preferences of said second community; and

determining said first community from said second community with members of said first community having a first preference in common so that said first community arises from a larger second community, said first community determined by having said first preference in common;

determining characteristics of said first preferences with regard to said first data streams to provide determined characteristics (930);

biasing an individual data stream arising from said first music-related database according to said determined characteristics so that said individual data stream is biased according to said determined characteristics and so that said individual data stream is biased for positive first preferences of said first community and biased against negative first preferences of said first community; and

transmitting said individual data stream (940) on a voluntary or selectable basis thereby allowing an individual to receive said individual data stream on a voluntary or selectable basis; whereby

said individual data stream has more content that said community likes and less content that said community dislikes.

STATEMENT UNDER ARTICLE 19(1)

Applicant respectfully submits the accompanying replacement sheets, pages 47 and 48, for the above-captioned international application. Claims 1, 3, and 6 have been amended without the introduction of new matter into the application. The amended claims now set forth the presence of a community with a plurality of members as well as the reliance upon a music-related database of songs and/or music videos.

Applicant believes that no impact is made upon the description and the drawings by the changes in the replacement sheets. Consequently, The art cited by the Search Report is not believed to adversely affect any of the outstanding claims, either amended or original.

If the Authority believes that a telephonic or other conference would be of value in expediting the examination and/or prosecution of the present application, Applicant invites the Authority to contact the undersigned counsel to arrange for such a conference.

With the above-referenced changes, it is believed that the application is in a condition for examination; and Applicant respectfully requests same.

PATENT
450117-05189

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicants:	Stefan MICHELITSCH et al.
Filed:	Filed Concurrently Herewith
Title of Invention:	CONTENT RECOMMENDATION DEVICE WITH USER FEEDBACK

745 Fifth Avenue
New York, New York 10151
Tel. (212) 588-0800

EXPRESS MAIL

Mailing Label Number EV468998740US
Date of Deposit June 4, 2004
I hereby certify that this paper or fee is being deposited with the United States Postal Service "Express Mail Post Office to Addressee" Service under 37 CFR 1.10 on the date indicated above and is addressed to the Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450

Charles Jackson
(Typed or printed name of person mailing paper or fee)

Charles Jackson
(Signature of person mailing paper or fee)

INFORMATION DISCLOSURE STATEMENT

Mail Stop Patent Application
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Sir:

Enclosed are copies of references which were cited in the search report of November 11, 2003 from the European Patent Office in connection with the corresponding European patent application. Also enclosed is a copy of Form PTO-1449. Entry of this Information Disclosure Statement and an early examination on the merits are respectfully solicited.

Please charge any additional fees to Deposit Account No. 50-0320.

Respectfully submitted,

FROMMER LAWRENCE & HAUG LLP
Attorneys for Applicants

By: /William S. Frommer/
William S. Frommer
Reg. No. 25,506
Tel. (212) 588-0800

Based on Form PTO-1449 (3/90)	ATTY. DOCKET NO. 450117-05189	SERIAL NO. Filed Concurrently Herewith
LIST OF REFERENCES CITED BY APPLICANT (Use several sheets if necessary)	APPLICANT Stefan MICHELITSCH et al.	
	FILING DATE Filed Concurrently Herewith	GROUP

U.S. PATENT DOCUMENTS

EXAMINER INITIAL		DOCUMENT NUMBER	DATE	NAME	CLASS	SUBCLASS	FILING DATE IF APPROPRIATE
	AA	5 790 102	08/04/98	Nassimi			
	AB	6 438 579	08/20/02	Hosken			
	AC	6 108 493	08/22/00	Miller et al.			
	AD	2003/051240	03/13/03	Schaffer et al.			
	AE	6 118 431	09/12/00	Terrier et al.			
	AF	5 571 997	11/05/96	Gray et al.			
	AG						
	AH						
	AI						
	AJ						
	AK						

FOREIGN PATENT DOCUMENTS

		DOCUMENT NUMBER	DATE	COUNTRY	CLASS	SUBCLASS	TRANSLATION YES	TRANSLATION NO
	AL	WO 01 15449	03/01/01	PCT				
	AM	WO 01 06398	01/25/01	PCT				
	AN							
	AO							
	AP							

OTHER PRIOR ART (Including Author, Title, Date, Pertinent Pages, Etc.)

	AR		Patent Abstracts of Japan, vol. 1998, no. 1, 30 January 1998, Publication No. 9244801, Publication Date September 19, 1997

EXAMINER	DATE CONSIDERED

* EXAMINER: Initial if reference considered, whether or not citation is in conformance with MPEP 609. Draw line through citation if not in conformance and not considered. Include copy of this form with next communication to applicant.

This Page Is Inserted by IFW Operations
and is not a part of the Official Record

BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images may include (but are not limited to):

- BLACK BORDERS
- TEXT CUT OFF AT TOP, BOTTOM OR SIDES
- FADED TEXT
- ILLEGIBLE TEXT
- SKEWED/SLANTED IMAGES
- COLORED PHOTOS
- BLACK OR VERY BLACK AND WHITE DARK PHOTOS
- GRAY SCALE DOCUMENTS

IMAGES ARE BEST AVAILABLE COPY.

As rescanning documents *will not* correct images, please do not report the images to the Image Problem Mailbox.